(1)

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91D-23

A Hybrid Approach to Battlefield Parallel Discrete Event Simulation

THESIS

Steven R. Soderholm
Captain, USAF

AFIT/GCS/ENG/91D-23

AFIT/GCS/ENG/91D-23

A Hybrid Approach to Battlefield Parallel Discrete Event Simulation

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Systems)

Steven R. Soderholm, BCS, MCIS
Captain, USAF

December, 1991

Approved for public release; distribution unlimited

Acknowledgments

I would like to thank my thesis advisor Major Eric Christensen for his wise counsel and guidance over the course of the past year. I'd also like to express my appreciation to my other committee members Dr. Thomas Hartrum and Major Michael Garrambone. Thanks to Mr. Rick Norris for going "above and beyond" in his efforts to keep the machines running.

My heartfelt thanks go out to my loving wife Lori, and my children Brian, Carrie, and Adam. Although the quest for knowledge never ceases, Lori won't have to say: "Leave Dad alone. He's studying" quite as often anymore.

Steven R. Soderholm

Table of Contents

Page

List of Figures

List of Tables

AFIT/GCS/ENG/91D-23

Abstract

This ~~study~~ Thesis describes a method of parallelizing a battlefield discrete event simulation. The method combines elements of conservative time synchronization together with elements of optimistic computation and local rollback on a message passing hardware architecture.

The battle simulation features aircraft moving in a battle area and launching missiles at enemy aircraft. Aircraft are randomly grouped into logical processes (LPs), and a single LP is assigned to each processor. Aircraft state information is replicated across all LPs. Only the LP with the minimum next event time can execute safely. While one LP is executing safely all other LPs are precomputing their next event. When an LP does become safe to execute it can update its previous precomputation and broadcast the results to all other LPs.

The sequential battlefield simulation has no battlefield partitions, and therefore the pros and cons of partitioning the battlefield in a conservative parallel implementation are discussed.

Simulation speedup was achieved without battlefield partitioning and various simulation scenarios were run in order to investigate the impact of event interleaving among logical processes on simulation speedup.

A Hybrid Approach
to
Battlefield Parallel Discrete Event Simulation

I. Introduction

August 1990 - The President asks our military leaders to determine how long it will take to have an effective defensive force in place in Saudi Arabia to deter Iraqi aggression, and immediately, operational planners set out to determine the impact of the mission through air mobility simulation. According to the Government Accounting Office (10:2,6) (11:2,10) military computer simulations are used throughout the Department of Defense to help decision makers answer tough questions like this. Unfortunately, such large and complex military computer simulations may take hours or days (12) to complete a run on single processor computers. In addition, typically in a stochastic simulation, multiple runs are required in order to reach stable parameters and to examine statistical variance on these measures.

In this fast changing unpredictable world, it's not always possible for our planners to anticipate these tough time-critical situations. To support quick planning and decision making during the unexpected crisis, we need to develop methods to speed up military computer simulations.

One approach to the problem of simulation speedup is to take the simulation that runs on a single processor and move it to a multiprocessor hardware architecture. This move allows parts of the simulation to run in parallel, and as one hopes, achieves significant simulation speedup.

1.1 Background

1.1.1 Computer Simulation.

It is often important to test a system to gain a better understanding of a system's effectiveness under various conditions. Yet it is often impractical or impossible to conduct the tests under *real world* conditions. The cost may be too great, the danger too great, or the system might be a proposed system and not even exist.

Computer simulation, also called computer modeling, is an attempt to translate the observable behavior of a real or proposed system into a computer program or computer model. The goal is to develop a computer model that represents the essence of the system and its environment accurately enough to provide results that can reliably predict the performance of the system.

The first step in translating the observable behavior of a system into a computer model is to identify the essential parameters of a system. In a complex system the modeler is not normally interested in the entire system, but some subset of the system. Therefore the essential parameters or the parameters of interest depend upon the nature of the system itself, and the goals of the modeler, and do not normally include every observable parameter in the actual system.

For instance, two separate computer programs may model an F-15 aircraft in flight. However one program is designed to provide insight into the flight dynamics of the aircraft and the other is designed to provide insight into the F-15's capability to detect, acquire, and destroy enemy aircraft. There may be some characteristics of the F-15 that are essential to both computer models such as maximum speed, but undoubtedly the flight dynamics model will painstakingly attempt to model the physical shape and aerodynamics of the F-15, while the search and destroy F-15 model is more interested in avionics and weapons systems characteristics, and may not consider the shape of the F-15 at all.

After the essential characteristics of the system are captured in the computer model, the simulation model is started up, and one can observe the behavior of the model over time. A good model will provide valuable insight into the behavior of the real or proposed system. However it should be understood that a model only attempts to predict performance.

For instance, to determine the effectiveness of the Patriot missile versus an incoming Scud missile, a modeler will first identify the essential observable parameters of the respective missiles, such as maximum speed, radar signature, and sensor capability. Next the modeler incorporates those features into a computer model. Finally, the modeler develops an input scenario that describes the initial state of the model. In this example the input scenario might consist of numbers and locations of Patriot missile batteries, and numbers, locations, and initial trajectories of Scud missiles. The output from the subsequent computer simulation runs helps the analyst determine the effectiveness of the Patriot against the Scud for particular attack scenarios.

1.1.2 **Simulation Resolution.** The resolution of a computer model refers to the level of detail incorporated into the into the model. The smallest entity modeled in a low resolution simulation might be an aircraft squadron in which case the observable parameters could consist of squadron morale, overall level of pilot experience, and numbers of aircraft. The outcome of a battle engagement between opposing squadrons might be estimated based on these parameters, and the result would be a decrease in the number of aircraft for one or both squadrons.

A high resolution simulation on the other hand might model a single dogfight between two opposing aircraft in a great amount of detail, incorporating observable parameters such as the flight characteristics, flight maneuvers, and missile or gun shots by each of the aircraft.

A simulation could contain a mixture of both high and low resolution modeling levels. As an example, the modeler may not be interested in the details of a training exercise and simply note that the training occurred by increasing the squadron's pilot experience level. On the other hand within the same simulation, aircraft attrition might be modeled through high resolution one-on-one dogfights.

In general the greater the amount of detail in a model the greater the computational complexity and intensity, and the greater time a simulation takes to complete.

1.1.3 **Simulation Types** There are two primary types of simulation models: continuous models and discrete event models.

According to Pritsker (24:58-59):

> In a continuous simulation model, the state of the system is represented by dependent variables which change continuously over time. A continuous simulation model is constructed by defining equations for a set of state variables whose dynamic behaviour simulates the real system.

Continuous simulations normally make extensive use of differential or difference equations to determine the value for the state variables at any given instant in time.

In a discrete event simulation the state of the system is updated at discrete instances in time, normally when some defined significant event occurs. Whereas with the continuous simulation system state is updated continuously over time. This research is concerned only with discrete event simulations.

1.1.4 **Simulation Time.** Simulation time refers to the time in the simulation and does not refer to wall clock time. For instance, a simulation of colliding atomic particles may be so complex that after running on the computer for several hours, only two seconds of simulation time have elapsed since the collision.

There are two primary methods of advancing the simulation time in a discrete event simulation. The first is called time driven simulation and the second is called event driven simulation.

1.1.5 Time Driven Simulation.

In a time driven simulation the simulation clock is advanced by some fixed time increment, and the state of all of the objects in the simulation are updated to correspond to the new time. In the Patriot vs SCUD example a new trajectory and location will be calculated for each of the incoming SCUD missiles and for each of the rising Patriot missiles at each tick of the simulation clock. If the simulation clock increment is too small then many calculations are done without anything significant happening. Going back to the SCUD example with a small time increment, on each tick of the clock the SCUD may only move several meters. If the SCUD is traveling thousands of meters then many location updates are calculated where the SCUD only moves a short distance on each update. On the other hand if the clock increment is too large, fewer calculations are needed, but significant events may get missed. In a large time increment simulation time may progress more quickly, and the SCUD may travel to its target location without as many location updates, but if we are also interested in the Patriot's target tracking and flight control mechanism we may lose track of many significant events since flight control commands may have been issued every tenth of a second.

1.1.6 Event Driven Simulation.

An Event Driven Simulation is usually referred to as a Discrete Event Simulation. Instead of updating the simulation clock a fixed time increment as in the the time driven simulation, event driven simulation updates the clock when significant events occur.

Going back to the Patriot vs SCUD example, significant events include the SCUD launch, the SCUD entering the Patriot battery's radar range, the SCUD reaching its target, the Patriot launch, and the Patriot destroying the SCUD. When the SCUD launch event is executed a future event or events are calculated for the

SCUD and these future events are inserted into a time ordered list of events called the Event List or Next Event List. After one event is executed the earliest or *next* event is taken off of the Next Event List and is executed. The simulation clock takes on the value of the time of the event being executed. The simulation clock does not advance in fixed time increments but jumps from one time to the next as events are taken off of the Event List and executed. The event execution itself can change the state of the system and/or schedule more events in the future. The time spent executing the event doesn't take any simulation time. The simulation essentially skips over the wasted calculations executed for the hundreds of miles where the SCUD is in flight where nothing significant happens.

1.2 Parallel Discrete Event Simulation

According to Fujimoto (7:19-20) the greatest opportunity for parallelism lies in executing events concurrently. The fact that the results of one event execution can impact other future events causes problems for concurrent execution. If one event can directly or indirectly impact another event then those two events must be executed sequentially no matter how many hardware processors are available.

1.3 Problem Statement

The challenge to parallel discrete event simulation is to take advantage of the parallelism inherent in the real process simulated by using multiple processors to execute independent events concurrently, and at the same time maintain simulation integrity by executing interdependent events in sequence. The difficulty lies in the fact that it is often hard to know apriori which events are independent and which events are interdependent.

1.4 Research Objective

My research goal was to take the discrete event battle simulation developed by Rizza (29) and parallelize it using a hybrid approach consisting of a variation of Chandy and Sherman's conditional event execution for conservative time synchronization, together with a variation of Reynold's optimistic local rollback.

1.5 Research Questions

1. Can battle simulation speedup be achieved using a parallel protocol hybrid model consisting of conservative time synchronization and optimistic next event lookahead on a parallel message passing hardware architecture?
2. Is there an effective method to execute in correct sequence, identical time events on separate processors?
3. How much does the parallel programmer need to know about specific details of the simulation application in order to parallelize the simulation?
4. How much does the application programmer need to know about parallel programming in order to facilitate parallelization of a sequential simulation?
5. How much of the parallel protocol design and code is reusable between different simulation applications?
6. Is it possible to develop and use a simulation testbed that features interchangeable simulation applications and interchangeable parallel time synchronization protocols? (related to #4 above)

1.6 Assumptions

- Since this battle simulation was developed in a short period of time by a novice modeller the assumption was that there were some deficiencies in the program. Initial testing of the simulation proved this assumption to be true. Several weeks were spent testing and debugging in order to bring the sequential battle simulation to the point where it would run to completion and provide correct results on a single processor.

- Elements of the SPECTRUM testbed (27:671-679) were used for this research, and there was expected to be a significant learning period associated with understanding the workings of the testbed (27:676). These expectations proved to be true. Additionally, modifications were made to elements of both SPECTRUM and the battle simulation in order to realize some compatibility. It was difficult to integrate all of the hybrid protocol with SPECTRUM and as a result only parts of the testbed were used during this research.

1.7 Scope

The primary goal was to parallelize the sequential battle simulation to achieve speedup while maintaining the integrity of the simulation, and no attempt was made to improve the battle simulation or make it more realistic.

1.8 Limitations

The characteristics of the battle simulation were not modified to facilitate parallelization in order to achieve speedup. For example, in the sequential battle simulation an event execution can result in the scheduling of another event at the current time or zero time increment. In order to improve the chances for parallel speedup one might change the original sequential battle simulation so that there is a minimal time increment greater than zero for event scheduling. The original battle simulation was not altered to make it more amenable to parallel speedup.

II. Literature Review

2.1 Introduction

This literature review focuses on parallel discrete event simulation. Most articles referred to logical and physical processes and therefore those concepts are described first. Within the realm of discrete event simulation there seem to be two major application types. These two types are explained together with some examples. As mentioned in Chapter 1, one of the primary challenges for parallel discrete event simulation speedup is how to maintain correct causal relationships between events executing on separate logical processes. In this regard there have been two primary areas of research. The conservative class of algorithms prevent causal errors from occurring, while the optimistic class of algorithms allow causal errors to occur and then execute some recovery or rollback scheme. The bulk of this review then looks at the research related to the conservative and optimistic algorithms of time synchronization for parallel discrete event simulation.

2.2 Parallel Decomposition

Chandy and Misra (4:198-199) describe a scheme where the real or proposed system being modeled is decomposed into physical processes. Each of these physical processes communicate with other physical processes in the system via messages. Logical processes are the abstract or computer representations of the physical process. Each physical process has an associated logical process. The logical process has input and output interfaces and performs some computation. Logical processes are assigned to multiple processors in a multiprocessor hardware architecture. Lee (16) mapped logical processes to processors in various configurations and determined that logical processes that communicate with one another should be mapped to the same processor in order to minimize inter-processor communications overhead.

2.3 Simulation Types

In the area of parallel discrete event simulation research the application types seemed to fall into two main categories. Nicol (22:1-3) describes the two types of models as message initiating and self initiating.

2.3.1 Message Initiating.

In the message initiating model each logical process re-evaluates its state (event execution) as a direct result of receiving a message from another logical process.

A network simulation is an example of a message initiating model. In a network simulation entities move through a system from one process or service activity to another (24:96-153). Upon arrival at a process the object may incur some waiting time and when the object gets serviced by the process, a service time may be incurred. One example is customers arriving, being serviced by, and departing a bank teller. Another example is a signal flowing through an electronic circuit. When the signal arrives at some electronic device it incurs some time delay before it is routed to the next device. State information associated with the entities such as signal strength or bank balance is maintained and updated as the simulation progresses.

In the message initiating model logical processes correspond to service activities and the entities traveling between service activities correspond to messages sent between logical processes. When a logical process receives a message from another logical process it updates or re-evaluates its state and sends the message on to another logical process.

2.3.1.1 Message Initiating Inherent Parallelism.

The parallelism inherent in the message initiating model lies in the fact that two logical processes that are not connected to one another can have no causal impact on each other and therefore can execute simultaneously. By "not connected" it is meant that the output from one logical process does not flow either directly or indirectly to another logical process.

2.3.2 **Self Initiating.** The self initiating logical process itself determines when it will re-evaluate its state. Although a logical process will still receive messages from other logical processes, these messages serve as update messages containing information that the receiving logical process will need when it eventually executes an event. A battlefield simulation is an example of a self initiating model.

In a battle simulation the modeller attempts to model the complexities of the battlefield environment. The battle simulation generally consists of objects such as infantrymen, vehicles or aircraft moving about a battlefield and interacting (attacking, evading, or coordinating) with one another. New objects such as missiles may be dynamically created, and objects such as the missile and its target may be dynamically destroyed. A colliding pool balls simulation and a colliding pucks (13:56-60) simulation are also self initiating models and pose some of the same problems for parallelization as those posed by the battlefield simulation.

In the self initiating model logical processes correspond to the tanks, aircraft, or pool balls. A tank may receive updated location information from another tank, but that information will not necessarily trigger an immediate event execution for the receiving tank. However, the receiving tank might make use of that information when it executes its next scheduled event, or it may use that information to schedule another event.

2.3.2.1 **Self Initiating Inherent Parallelism.** The parallelism inherent in a self initiating model is more difficult to isolate than in the message initiating model. Although potentially any object can impact any other object, most rely on the fact that as the distance between objects increases the possibility of causal dependencies between the objects decreases. Therefore the inherent parallelism is based on the distance between objects. As the distance increases the amount of parallelism possible increases.

2.4 Time Synchronization

Time synchronization research revolves around finding efficient methods of keeping all of the logical processes in time synchronization to ensure that the correct sequence of interdependent event executions are maintained throughout the simulation. The methods of doing this fall into two major categories: conservative and optimistic.

2.4.1 **Conservative.** The conservative class of algorithms, initially developed by Chandy and Misra (3:440-452), and Bryant (2), force each of the processors in a multiprocessor parallel simulation to "play it safe." These algorithms are designed to prevent cause-effect violations. In this scheme each logical process runs its own part of the simulation and transmits results of its computations in the form of messages to other logical processes. Message timestamps are always increasing. In Fig 2.1 logical process (LP) #1 received a message at time five. LP #1 calculated a service time of five and sent messages to LP #2 and LP #3 with a timestamp of ten. LPs #2 and #3 realizing that they can receive no message with a timestamp earlier than ten, can process events concurrently up to a time of ten. LPs #2 and #3 both have service times of five and six respectively, and they each send a message to LP #4 with a timestamp of 15 and 16 respectively.

When each logical process looks at its set of incoming message timestamps it identifies the earliest or smallest incoming time. The smallest incoming message time is its safe time. It then executes events in its next event list which have a time less than or equal to the safe time. In this case LP #4 can execute events up to a time of fifteen. The logical process then blocks until it receives a new minimum

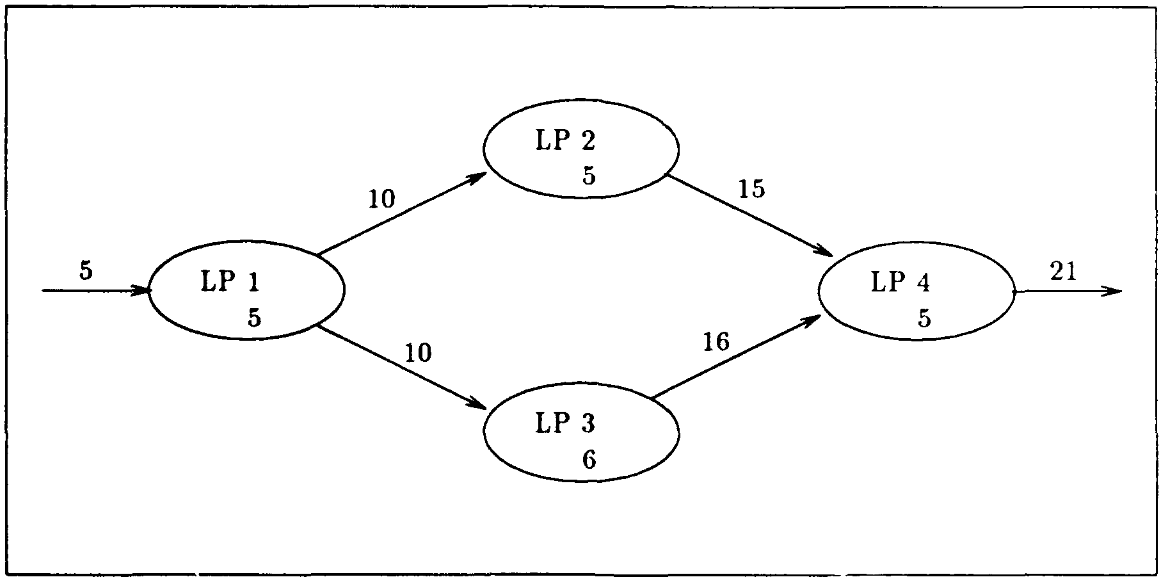

Figure 2.1. Conservative

safe time. Unfortunately, in this scheme it is possible to enter the deadlocked state where there is a cycle of logical processes all waiting to receive new safe times.

For example (Fig. 2.2), if LP #1 sends a message timestamped at time ten to LP #2 but not to LP #3, then LP #3 cannot process any events and neither can LP #4. If the output of LP #4 somehow feeds back into LP #1 then deadlock exists and no more events will be processed in this part of the network.

2.4.1.1 **Deadlock Prevention.** Much of the research to date has revolved around avoiding and recovering from deadlock. In order to avoid deadlock Chandy and Misra (4:210-202) developed a scheme of sending null messages containing time updates to prevent deadlock. In the example demonstrated with Fig 2.2 when the message with a timestamp of ten is sent to LP #2, a null message with a timestamp of ten is sent to LP #3. LP #3 then adds its minimum service time of six to the incoming null message time and sends a null message with a timestamp

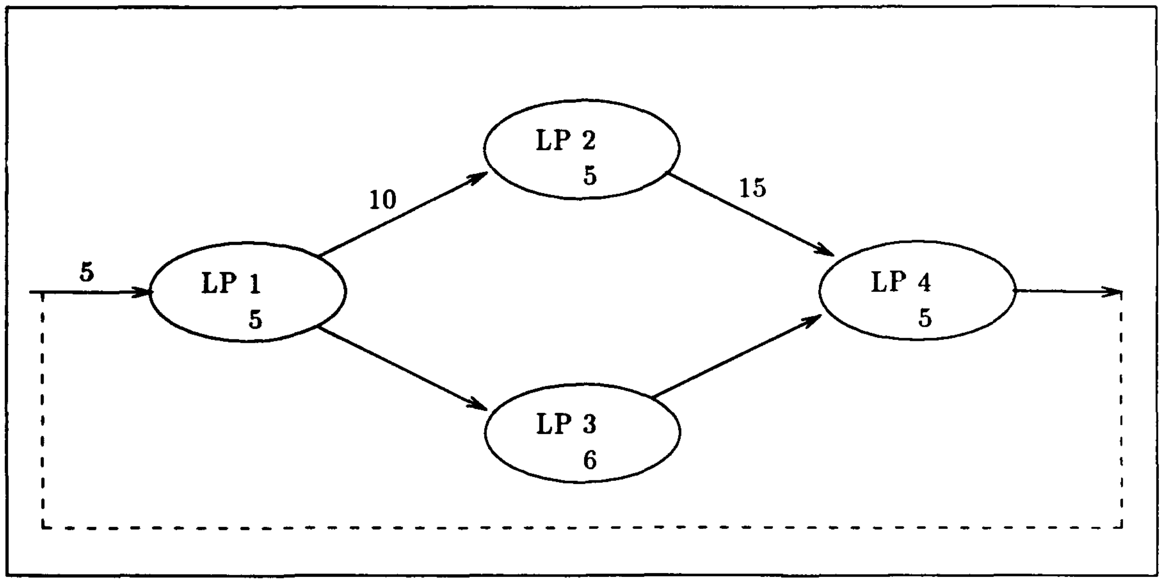

Figure 2.2. Conservative - Deadlock

of 16 to LP #4. Notice that in this case the service times at each LP represent the minimum service times for any incoming message.

Nicols and Reynolds (23:471-474) propose a variation on the null message passing scheme called SRADS whereby instead of automatically sending a null message for each received message, a logical process will send a null message when it receives a request for one from another logical process. A logical process will send a request for a null message when it detects that it is about to block waiting for a time update from another logical process.

2.4.1.2 **Deadlock Detection.** Another approach allows deadlock to occur, detects the deadlock, and breaks the deadlock. Several deadlock detection techniques have been proposed. Misra (19:62-63) proposes one where a "marker" circulates among logical processes marking those that are blocked. If the marker detects a cycle of blocked logical processes it raises the deadlock flag, and then time updates, which effectively break the deadlock, are distributed to the logical processes.

2.4.1.3 **Time Windows.** Another method of conservative time synchronization called Time Windows by Lubachevsky (17) sets up a time window or time range for which it is safe to process events. This range is dependent upon prior knowledge of minimal increments between event times.

2.4.1.4 **Pre-Computing.** Nicol (21) proposes a scheme of precomputing events in the next event queue to gain better future knowledge of events that are safe to process.

2.4.1.5 **Conditional Events** Chandy and Sherman (5:93-99) propose the conditional event approach to discrete event simulation. In this paradigm events that are safe to process are called definite events, and events that are not guaranteed to be safe to process are called conditional events. The algorithm for this method is as follows:

loop

Each logical process:

1. determines if it has definite events. If a logical process's next event has the minimum time of all logical process's next events then the logical process has a definite event.
2. executes or computes as many definite events as possible.
3. sends the time of its pending conditional event after having computed all possible definite events.
4. waits to receive a message from each logical process. These messages contain the pending conditional event time. These times are stored and facilitate the computations in step 1.

end loop.

2.4.2 **Lookahead** Lookahead refers to the ability of a logical process to accurately predict when it will send out its next message to another logical process. Nicol (22:4) classifies two types of lookahead: time-lookahead and full-lookahead. If a logical process has time-lookahead it can predict the simulation time in its next output message before the next event in the next event list is executed. Full-lookahead means that not only can the logical process predict what time the message will be issued but also what the contents of the message will be. Lookahead is based on some knowledge of the physical process that is being simulated such as the minimum service time at a bank teller window or the time delay of an electronic signal as it passes through a transistor.

Fujimoto (7:23) states that conservative algorithms must be "adept at predicting what will not happen". That is to say, in order for two logical processes to execute simultaneously, with different local simulation time, each must be sure that it will not receive a message in the past from the other logical process. According to Fujimoto (7:23) there are three aspects of an event driven simulation that impact this ability to predict:

1. Lookahead - some predetermined knowledge of the simulation application that allows a logical process to predict its future message time and pass that information on to other logical processes.
2. Event Message - the receipt of a message guarantees the receiving logical process that it will receive no earlier messages from the sending logical process.
3. Simulation Structure - two logical processes that are not connected either directly or indirectly can execute events without regard to each other.

If a simulation has no ability to predict then only the logical process that contains the event with the smallest timestamp is safe to execute, since that event could potentially impact any other pending event on any other logical process.

2.4.3 Conservative Performance.

Most of the research to date on the conservative time synchronization approach has been with queuing network simulations (message initiating).

Fujimoto demonstrated (8) that lookahead is indeed critical for achieving speedup in parallel network queuing simulations.

Su and Seitz (31) achieved some speedup with logic simulations and a deadlock avoidance algorithm.

Reynolds (27:675) had difficulty when attempting to use the null message protocol with a battle simulator. The failure was due to zero time increment for time-lookahead.

Chandy and Sherman (5:97-98) report some speedup with the conditional event approach and a queuing network simulation. They predict that their algorithm will work well on simulations such as circuit simulations, but expect poor performance on simulation problems that have a fully connected structure of logical processes.

Reed and Malony (26:11) conducted research with queuing network simulations on a shared memory architecture. They used both Chandy and Misra's conservative deadlock avoidance and deadlock detection techniques and conclude that those techniques are not well suited for queuing network simulations. They go on to state that it is unlikely that the Chandy-Misra technique is well suited for any type of simulation model.

2.4.4 Optimistic

Instead of playing it safe, the optimistic class of algorithms make no effort to prevent cause-effect violations.

The most studied optimistic scheme is called Time Warp and is based on Jefferson's (14) paradigm of virtual time. In Time Warp each logical process executes freely without regard to what other logical processes are doing. But if a logical process receives a message "in the past" from another logical process. Then the receiving logical process must rollback its state to a time less than or equal to the time of the received message. This also implies that events will be re-executed and cancellation messages will be transmitted to negate previously sent messages. The challenge in this paradigm is to first determine that a message in the past has been received, and then system-wide, cancel erroneous events, move back to safe states, and continue the simulation.

To facilitate rollback, each logical process must save previous states. Due to limited memory, previous states cannot be saved indefinitely. Therefore periodically the simulation must come to a standstill while global virtual time is calculated. Global virtual time is the minimum system wide safe time. That is to say, no logical process will rollback to a time earlier than the global virtual time. After global virtual time is calculated and distributed to each logical process, each logical process can free the memory containing state information with times less than global virtual time.

Variations of the optimistic approach revolve around optimizing the rollback mechanisms. One such method called lazy cancellation (9) attempts to determine the extent of the harm done by the message from the past. If the output messages from the first "erroneous" execution are the same as generated from the "correct" execution then cancellation messages are not sent and the rollback is complete. Otherwise a normal rollback complete with cancellation messages is executed.

Other variations such as Moving Time Windows (30:34-42) attempt to put an upper limit on how far any given logical process can "get ahead" of the other logical processes. This scheme is based on the premise that the logical process that is farther ahead in time than the other logical processes is the one most susceptible to receiving a message in the past from one of the other logical processes.

2.4.5 **Optimistic Performance** Jefferson (15:741) reported a speedup of 10.5 on the Jet Propulsion Laboratory's 24 node Mark III Hypercube using Time Warp with a pool balls simulation. Wieland (32:1273-1274) reported a speedup of 4.5

on a 32 node Mark III Hypercube with a division on division land combat simulation. Wieland subsequently artificially increased the computation time in relationship to the message passing time and observed an increase in speedup from 4.5 to 12.

2.4.6 **Conservative versus Optimistic.** Both paradigms have their proponents, and both paradigms have their weaknesses. Fujimoto (7:23,26) summarizes the weaknesses of both schemes:

Conservative Weaknesses:

- Application Parallelism not Fully Exploitable - Even if there is the possibility (no matter how slight) of one logical process either directly or indirectly impacting another logical process then those logical processes must execute in sequential order. Logical processes that could execute concurrently most or all of the time are forced into sequential execution and therefore opportunity for parallel speedup is lost.
- Speedup Dependent on Application Detail - To facilitate lookahead application detail must be known. Attributes such as minimum timestamp increments or the interconnection between logical processes must be programmed into the conservative protocol. Also changes in these attributes can significantly impact performance. According to Fujimoto (7:24) "it has been observed that seemingly minor changes to the application may have a catastrophic effect on performance."
- Static Configurations Required - All logical processes and their inter-connections must be statically defined. Techniques to circumvent this problem lead to excessive overhead.

Optimistic Weaknesses:

- Thrashing - Critics contend that the optimistic paradigm is susceptible to thrashing, where logical processes spend most of their time rolling back to previous safe states. There is no analytical proof to suggest that the optimistic paradigm is stable and will not succumb to thrashing. On the other hand empirical data to date indicates that thrashing is not a problem. Also, Jefferson (15) states that if there are no zero time cycles progress is guaranteed no matter how many rollbacks are accomplished.

- Memory Usage - Since the optimistic scheme must save state following each event execution there is increased memory usage in order to hold all of the *safe* state information. If the state saving requirements are large then there is the possibility of significant overhead associated with managing the state saving activities.

Fujimoto (7:26-27) concludes by pointing out that both techniques, although strapped with some weaknesses, have achieved some success. The optimistic paradigm however has achieved success across a broader range of application types than the conservative class of algorithms.

2.4.7 **Hybrid (Conservative with Optimistic)** Dickens and Reynolds (6:162-164) mention a technique called Local Rollback which combines the conservative with the optimistic method. In the Local Rollback paradigm while a logical process is waiting to become safe to execute it precomputes future events and saves the results of those computations but does not send the results to other logical processes. Once this logical process does become safe to execute it has a head start on its computations. If it receives a message in the past then a rollback mechanism must be invoked but no cancellation messages need be sent.

2.5 Hardware Enhancements

All of the time synchronization methods require some sort of global time synchronization between logical processes. Reynolds (28) presents a framework for hardware development of auxiliary parallel reduction networks. This hardware can rapidly transmit, receive, and process timing information which can be used for global synchronization calculations in any protocol.

2.6 Time Driven

Nicol believes (20:141) that neither the optimistic nor the conservative approach will be successful when used to parallelize a complex military simulation. His premise is that because of the large number of interdependencies in a military simulation, and the feeble lookahead capability, the conservative approach will get bogged down in deadlock detection and recovery, and the optimistic approach will spend all of its time rolling back to previous states. Nicol proposes that the time driven paradigm will be most effective for military simulations. In this model a global clock is used. The clock is incremented, then all processors process events up to the new clock time and wait for the next clock increment.

2.7 Separate and Simultaneous

Fujimoto (7:19) and others point out that there is a class of simulations involving stochastic processes. This includes most battlefield simulations. With this class, many simulation runs are made in order to minimize statistical variance. In these cases it is more effective to put a single copy of the simulation program on each processor in a multiprocessor architecture, and run them all separately and simultaneously. For example, using a single processor, a two hour stochastic simulation that needed 30 runs to minimize variance would take 60 hours to finish. On a 30 node multiprocessor architecture with a separate copy of the simulation on each node, 30 runs of the simulation would be complete in two hours.

2.8 Simulation Testbed

Reynolds (27) developed the SPECTRUM simulation testbed. The SPECTRUM testbed purportedly supports easy mixing and matching of simulation applications and parallel time synchronization protocols. Reynolds indicates that more was learned about parallel simulations doing the actual protocol and application development then by running experiments to determine speedup. Specifically he had underestimated the amount of application specific knowledge required to implement an application with a specific protocol. He outlined the need to identify classes of simulation applications and classes of parallel protocols and then determine which applications fit best with which protocols.

2.9 AFIT Research

Proicou (25) used the Chandy-Misra algorithm with independent logical processes in a network queuing type of simulation and mapped logical processes to physical nodes in an arbitrary fashion. He achieved no speedup. Mannix (18) embedded an event list in every logical process and claims to have achieved superlinear speedup when no feedback loops were present. Lee (16) continued on with Mannix's research and combined logical processes with the hope of reducing interprocess and therefore interprocessor communication. Lee could not validate Mannix's superlinear speedup. Proicou and Lee used the SPECTRUM testbed which provides for a suite of time synchronization protocols which are supposedly transparent to the application programmer. In each case SPECTRUM was not transparent to the application, and SPECTRUM and/or the application had to be modified to improve compatibitily.

2.10 Summary

Conservative and optimistic time synchronization are the two main areas of parallel discrete event simulation research.

In the conservative approach safe events are executed. Safe events are events that can be executed with the guarantee that a logical process will not receive a message with a timestamp earlier than the time in the safe event. Following safe event execution, time checking is accomplished between connected logical processes to determine which events are safe to execute on the next iteration.

In the optimistic approach logical processes execute events freely without regard to which events are safe to process. However if a logical process receives a message in the past then it must rollback by returning to a safe state. This implies state saving and recovery, and cancellation message transmission.

A hybrid approach which combines conservative time synchronization with local rollback has been attempted, but to date no results have been published.

Hardware support for rapidly disseminating and processing synchronization information holds promise for parallel discrete event simulation speedup.

A parallel simulation testbed (SPECTRUM) which supplies a generic set of parallel synchronization protocols that can be used with various applications has run into problems caused by application specific requirements which impact the ability of the testbed to be generic.

2.11 Conclusions

With the conservative approach to time synchronization there doesn't appear to be much research in the area of parallelizing battle simulations. Most of the research to date involves looking for parallel speedup of network queuing models. Battle simulations typically have poor lookahead capability and that makes them poor candidates for the conservative approach to time synchronization. Some empirical data has been collected using the optimistic approach with a battle simulation. The results of those tests indicate that it is difficult to achieve significant speedup with a battle simulation. The Spectrum generic testbed was developed with the hopes of facilitating protocol comparisons. AFIT researchers have had to customize the testbed to support particular applications thereby defeating the purpose of the testbed.

III. The Model and the Hardware

3.1 Overview

The AFIT Battle Simulation (RIZSIM) is a high resolution battle simulation model developed by Capt. Robert Rizza as part of his Master's Degree thesis efforts (29) at the Air Force Institute of Technology School of Engineering. This model is one of scripted movement of objects (aircraft, land vehicles, or missiles) in empty space. Each object continues on its scripted path unless it senses another object, in which case it either evades the other object, attacks it, or does nothing and continues on its path. Aircraft and land vehicles can be dynamically destroyed and removed from the simulation and missile objects are dynamically created. No aspect of the battle field environment such as terrain or weather was modeled.

3.2 Classifying the Model

The method of classifying this model was based on a Military Operations Research Society workshop report called SIMTAX - A Taxonomy for Warfare Simulation (1).

3.2.1 **Classification by Purpose.** The purpose of creating this model was to provide a "representative" military high resolution model that can be ported to the Air Force Institute of Technology's parallel computers for follow on studies and research into parallel event simulation. On the parallel architecture this model will be used as an analytical model in the area of developing and analyzing methods of parallel event driven simulations. In a sense, the reason for developing this model was to provide a model of a high resolution model. Some of the stated objectives and requirements of the model are as follows:

- should be a "representative" event driven military model.

- should be written in the 'C' programming language.
- the software should be easily maintained.
- should use object oriented design.
- computationally intensive.
- should provide output to a display driver.
- parallel processing issues should be considered during all phases of design and implementation.

3.2.2 Classification by Qualities.

- Domain - In this model the objects operate in three-dimensional space using X, Y, and Z coordinates. Although in the model itself the objects move about in empty space, the display driver shows aircraft and vehicles on land.
- Span - The span is virtually unlimited. The movement range of the individual objects is limited only by the maximum size of the coordinates (double floating point numbers).
- Environment - There is no terrain, weather, day/night, or cultural features. In short, the environment is not modeled.
- Force Composition - This model provides five different object or entity types: Mig and F-16 aircraft, tank and truck vehicles, and a missile. The missile is the only weapon type that this model currently supports. Other objects (but not weapon types) could be scripted but these are the object types that the graphical interface understands. Each entity has a loyalty to a given side, but organizational structures, C3, intelligence, and logistics are not modeled.
- Scope of Conflict - Conventional.
- Mission Area - The mission areas that it most closely matches from an Air Force point of view are either Armed Reconnaissance or an aircraft raid on a

target. Various other missions can be scripted, but due to lack of C3, and lack of flexibility in movement it is tough to model a mission more complex than that of a lone gun moving on or above the battle area looking for a target of opportunity or a pre-specified target. By lack of flexibility of movement I mean that, for example, an aircraft will stay on its prescript flight path and launch missiles towards any targets that it senses but it will not alter its flight path to 'go after' targets.

- Level of Detail - The detail is at the individual entity level. All entities operate (movement, searching, engaging) as individuals. Attrition processing is on a missile shot by missile shot basis.

3.2.3 Classification by Construction.

- Human Participation - Not Required/Not Permitted.
- Time Processing - Dynamic/Event Step.
- Treatment of Randomness - Random/Pseudo Random numbers not used. Each run of a given scripted scenario will produce identical results. This is a deterministic model of a stochastic process.
- Sidedness - One to many sided. All sides behave the same but in the scenario one side can be given greater capabilities than the other side. Objects of different loyalties are enemies. Since Ordnance Reached Target is a scheduled event in time, it is possible for a target to launch a missile after being fired upon but before being hit. Therefore this model could be considered to be a many-sided symmetric model.

3.2.4 **Hardware Requirements.** The software was written in standard ANSI 'C' and consists of approximately 1300 lines of code. The model was developed on a personal computer and then with some difficulty ported to a SUN workstation. Apparently the PC 'C' runtime environment handles pointers a little differently than

UNIX on the SUN. Some pointer errors not caught by the PC resulted in runtime errors on the SUN. The problems were fixed and at this time the source code can be compiled and run on virtually any type of computer

3.2.5 Input.

- Scenario - There are no scenario generation tools or preprocesses associated with this model. All of the attributes for each object (except missiles), and the route point data which controls the movement of the objects must be manually input. An input file is built by the scenario scriptor person and this file is read into the model at the beginning of the simulation run.
- Data - There is no terrain data. There is no lethality or attrition modeling and therefore no data associated with those areas. There are object attribute types available in the model such as aircraft performance and sensor and defensive system capability. These attribute types are not used. Which is to say that a scriptor could provide input aircraft performance information to the model, but the model would not make use of or process that information. There is no attempt to model a particular type of weapon system or avionics capability and hence no data requirements for those attributes.
- Instructions - There are no instructions associated with this model.

3.2.6 **Firing.** Each object has an unlimited number of missiles, so every target that is acquired will be fired upon. Objects in this model can fire missiles only. The missiles have no sensor capability. Each object has an unlimited number of missiles.

3.2.7 Engagement Assessment.

- Engagement Geometry - Engagement Geometry is not modeled. A missile can be launched in any direction. The location of the target in relationship to the attacking object has no impact on the probability of kill.
- Damage Assessment - Single round accuracy, single round lethality, and multiple round assessment models are not used. When a missile is launched at a target, the missile schedules a route point for the target's current location followed by the target's next routepoint. If the target reaches its next route point and heads off toward another routepoint before the missile reaches the target then the target escapes. Otherwise, the target is killed. Partially damaged objects are not modeled.

3.2.8 **Output.** Since there is no randomness in this model each given scenario will run exactly the same each time. Consequently there is no post processing done to accumulate and compare results of various runs. While the scenario is running significant event information is sent to a graphical output file. This file is used to generate a graphical animation of the scenario. The graphics software allows the user to view the battle from any perspective while the battle is being played back.

3.2.9 **Measures of Effectiveness.** This model does not compute or keep track of any measures of effectiveness. There are no killer-victim scoreboards or any other measure of success for a given scenario or mission.

3.3 The Model in Detail

3.3.1 **The Players.** There is a generic object type, from which scenario scriptors can build various types of objects such as aircraft, tank, truck, etc. Each object has the following attributes:

- Object Type - Icon identifier for interface to the graphics driver.
- Object Identifier - Unique number assigned to each object in the scenario.

- Object Loyalty - A number. Objects with the same number are loyal to each other.
- Current Time.
- Fuel Status.
- Condition - Value Range - 0 - 100. 0 – Destroyed. 100 – Fully operational.
- Vulnerability - Number indicating the destructive force needed to destroy the object.
- Location - In X,Y,Z coordinates.
- Velocity - X,Y,Z velocity vectors.
- Orientation - Yaw, Pitch, Roll of the object.
- Rotation Rates - of the object about X,Y,Z axis.
- Operator - Experience and Threat Knowledge.
- Performance Characteristics - Max Speed, Minimum Turn Radius, Average Fuel Consumption Rate, Max Climb Rate.
- Route Data - Linked list of scripted route points.
- Sensors - Linked list of sensors containing sensor type, range, and resolution for each sensor.
- Armaments - Linked list of armaments containing armament type, range, lethality, accuracy, speed, and count for each armament.
- Defensive Systems - Linked list of defensive systems containing defensive system type, range, and effectiveness for each defensive system.
- Targets - Linked list of scripted target types and locations.

Scenario scriptors can build various types of objects by varying the values for the different attributes listed above. The model itself, only makes use of or currently supports the following attributes:

- Object Type.
- Object Identifier.
- Object Loyalty.
- Current Time.
- Location.
- Velocity.
- Orientation (excluding roll).
- Route Data.
- Sensors (range only)
- Armaments (Missiles only)
- Targets (type only).

3.3.2 **Movement (update_position).** Movement for each object is controlled through the use of scripted route points. Each object has its own time ordered queue of routepoints. This queue of routepoints in essence defines each objects plan of movement or battle plan for the entire simulation. It is not possible for an object to increase or decrease its speed during the simulation run at the current time.

The first thing that happens with any event is an update position. The update position takes the next route point off of the route point queue and moves the object to that point. It then examines the next route point on the queue and adjusts the current velocity vectors for the object to point the object in the right direction toward the next route point. Naturally, prescript route points will be deleted if an object is destroyed during the run, but normally the prescript route points will not be removed or adjusted. However, route points can be added as the result of an object attempting to evade another object, or when an object moves into the sensor zone of another object (Entered Sensor Range) or when an object senses another object (Made Sensor Contact). There is a case where a routepoint will be skipped, and that is if the object is trying to evade another object that is 'camped out' close to its next route point.

3.3.3 Searching/Sensing (sensor_check).

After an object's position is updated, a sensor check is accomplished. The Sensor Check function is the behind the scenes master controller of this model. After the initial Reached Turnpoint events are scheduled for each object, the Sensor Check function takes over and schedules all subsequent events except for Ordnance Released.

The sensor check for each object can be thought of in the following way. Each object has a cylindrical sensor zone with the object in the center. Whenever Sensor Check is executed for a particular object as the result of an event execution, this sensor zone is swept from this primary sensing object's (PSO) current location to its next route point. One by one the path of movement for every other object is then projected to see if it will intersect the sensor zone of the PSO. The earliest time of intersection between the PSO and all of the other objects is kept as the projected event time for a future event. If an object has more than one sensor then the sensor with the maximum range is always used. Also, a check is made to see if the PSO will enter the sensor zones of any of the other objects in the scenario. So the Sensor Check is like a projection in time for an object out to its next route point. It will either sense some other object, be sensed by some other object or there will be no contact. The sensor algorithm is based on an object's movement between its current location and its next route point.

If an object penetrates the outer boundary of a sensor zone of another object there is a 100% chance that object will be detected. In this model there is no way for an object to suddenly appear inside of another object's sensor zone. That is to say, sensor contact is made only when an object intersects with the outer boundary of someone else's sensor zone. The scenario script can initially put one object inside

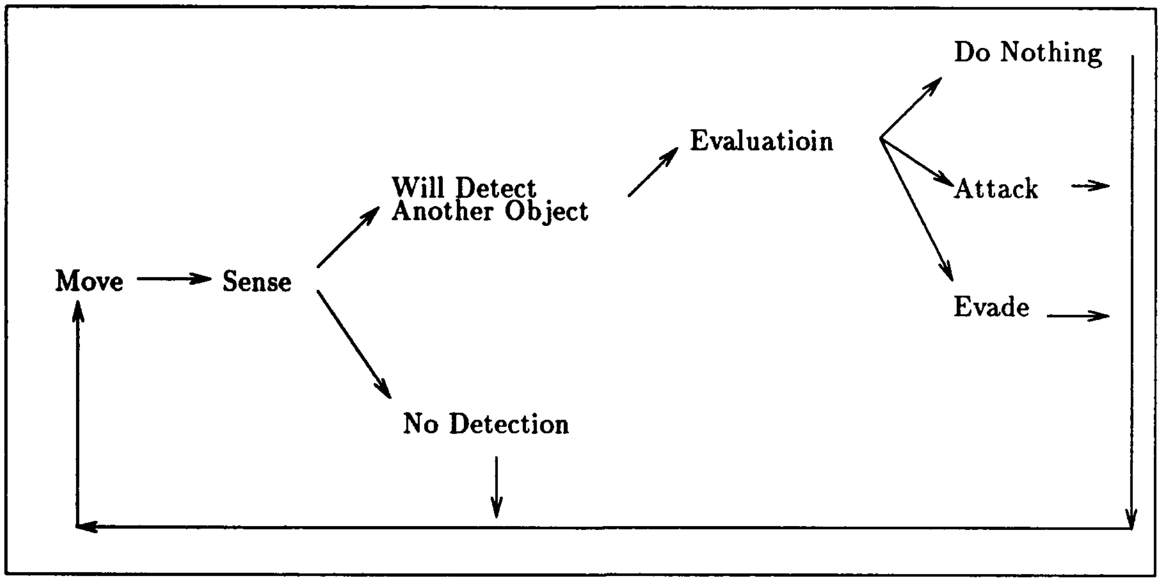

Figure 3.1. "Flow" of the Model

of another object's sensor range in which case there is no Made Sensor Contact. Unrealistically, an object that is inside of another object's sensor zone is ignored.

There is no randomness involved in this search model. There is a call to a line of sight check but that function always returns TRUE.

This algorithm only works for objects that are moving between route points. If an object is not moving it has no sensing capabilities.

After an object reaches its last scripted route point it no longer moves and therefore loses its capability to sense. For this reason, this model does not support stationary objects.

3.3.4 **Decision Making (sensor_check).** The decision process for scheduling future events based on sensor contacts for the primary sensing object is as follows:

1. If the PSO does not sense any other object and no other object will sense the PSO then schedule a Reached Turnpoint (advance to next route point) for the PSO.
2. If the PSO senses another object then schedule a Made Sensor Contact with the PSO and that object. If more than one contact will be made then schedule the earliest contact as the next event for the PSO.
3. If another object senses the PSO then schedule an Entered Sensor Range for the PSO. Earliest contact time applies here also.
4. If the PSO is a missile and senses a target then schedule an Ordnance Reached Target event for the PSO and the target. If a missile senses another object then it has collided with the object since the missile's sensor range is zero.

3.3.5 **Decision Making (operator_evaluation).** This evaluation comes about as a result of the Sensor Check function scheduling a Made Sensor Contact event. When Made Sensor Contact is executed it calls Operator Evaluation to determine what to do with this contact. Based on object loyalties, types on the target list, and sensor ranges, the object will either attack, evade, or do nothing. The target list is not dynamic it is prescript for each object and contains a list of object types that an object wants to attack.

3.3.6 **Target Acquisition (attack).** Objects that intersect other object's sensor boundaries will be detected with 100% probability. If the detected object is not the same loyalty as the PSO and it is one of the object types on the prescript target list then it will be selected as a target with 100% probability.

3.3.7 **Evasive Measures (evade).** If a PSO senses another object, of different loyalty, not on the target list, and with a smaller sensor range then the PSO evades the detected object. The evasion takes place by moving (setting up a route point) that is at 90 degrees away from the direction of movement of the detected object. This is an evasive maneuver to avoid detection. There are no evasive maneuvers to avoid enemy weapon's systems.

3.3.8 **Termination.** The scenario terminates when there are no more events on the event list. That is to say all of the objects have reached their final route point. The simulation driver provides a function that can be used to end the scenario before the event list is empty. That function is not used in this model.

3.3.9 **The Events.**

- Reached Turnpoint - This event is the initial event scheduled for all objects at the beginning of the simulation. This event is also scheduled by the Sensor Check function once the scenario is running. When the scenario is running this is the default event that gets scheduled if no other event applies. When Reached Turnpoint executes it calls two functions: update position and sensor check.
- Entered Sensor Range - When an object performs a Sensor Check the white player or master controller takes over in the form of the Sensor Check function. This function not only determines if this object will be able to sense anything between its current location and the next route point, it also determines if any other object will be able to sense this object. If some other object will be able to sense this object then an Entered Sensor Range event is scheduled and an intermediate route point is added at the point where the object will enter the other's sensor range. When the Entered Sensor Range event executes it calls two functions: update_position and sensor_check.
- Made Sensor Contact - When Sensor Check determines that this object will make sensor contact with another object between its current location and the next route point it adds an intermediate route point where sensor contact will occur and schedules Made Sensor Contact. When Made Sensor Contact

executes it calls three functions: update_position, operator_evaluation, and sensor_check.

- Ordnance Released - If the decision within Made Sensor Contact's operator_evaluation is to attack, a missile object is created and an Ordnance Released event is scheduled for the missile. When this missile is created its initial route point is the location of the object that released the missile, the next route point is the target's current location. The missile's final routepoint is the target's next routepoint. When the Ordnance Released event is executed it calls two functions: update_position and sensor_check.
- Ordnance Reached Target - If a missile object is executing a sensor check and the sensor check reveals a contact for the missile then the contact means that the missile has collided with the object (the sensor range for a missile is preset to zero) and an Ordnance Reached Target event is scheduled for the missile. When an Ordnance Reached Target event executes it calls the appropriate functions to delete the missile object and its target from the scenario if a hit is verified, otherwise, in the case of a miss, only the missile is deleted from the scenario. After viewing numerous simulation runs it is apparent that the hit_miss processing within Ordnance Reached Target always results in a hit. The real hit-miss determination is made by sensor_check. If the missile comes close enough to the target for sensor_check to schedule an Ordnance Reached Target then a missile hit will ensue.
- Collision Distance Reached - An analysis of the source code indicates that it is not possible to schedule this event.

3.4 RIZSIM Example Event Execution

Sequence of operations:

- Pop the next event off of the next event list.

- It is a Reached Turnpoint event for object #1.
- Update Position performs the following functions:

    1. Set the simulation time equal to the time of this event.
    2. Pop the next routepoint off of object #1's routepoint list.
    3. Update object #1's state information to reflect location at the routepoint.
    4. Look at the next routepoint on the routepoint list to determine heading toward the next routepoint from this location.

- Sensor Check performs the following functions:

    1. Sensor check determines that object #1 will make sensor contact with object #2 at some point between its current location and the next routepoint.
    2. Sensor check pushes a new routepoint onto object #1's routepoint list. This routepoint is at the point where object #1 will sense object #2.
    3. Sensor check schedules a Made Sensor Contact event and inserts it into the next event list. The time on the Made Sensor Contact event is the time at which object #1 will sense object #2.
    4. At this point the processing for the Reached Turnpoint event is complete and the simulation driver gets the next event from the next event list.

3.5 RIZSIM Bugs

Testing RIZSIM with random scenarios revealed problems in the software that resulted in software failure. Appendix A details the failures and the fixes or patches implemented to resolve the problems.

With the fixes in place the simulation works correctly with most random scenarios.

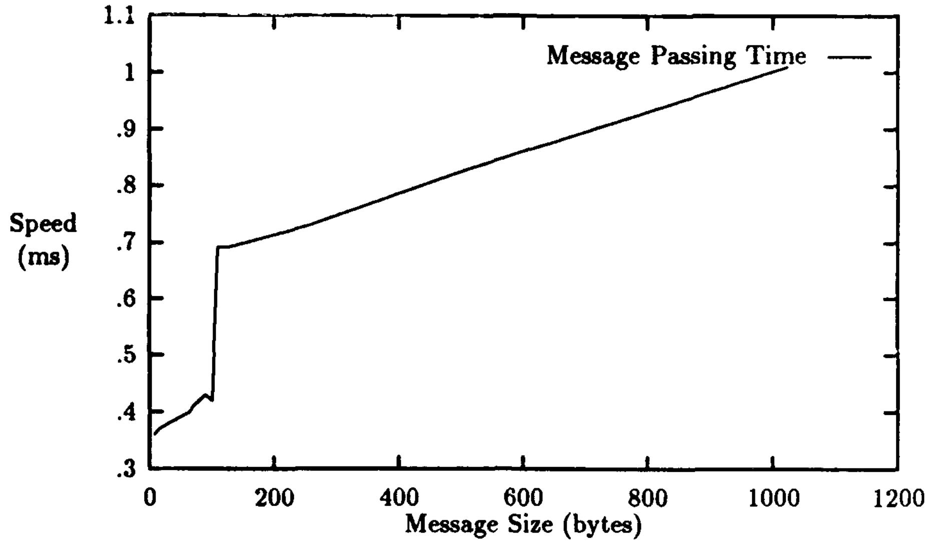

Figure 3.2. Hypercube Message Passing Time (millisecs)

Of note, the RIZSIM evade algorithm was not tested nor is it exercised by any of the scenarios.

3.6 The Hardware

The target machine for parallelization of this model is an Intel iPSC/2 eight node Hypercube. Each node consists of an 80386 microprocessor with an 80387 math coprocessor. This machine features Direct Connect Modules at each processing node. These modules basically support message passing that does not interrupt node processing.

The message passing time is based on the availability of a path between the sender and receiver and the size of the message. The chart in Figure 3.2 shows the message passing time between nodes. This chart was derived from message passing experiments with an eight node Hypercube configured as a ring.

IV. Parallel Considerations

4.1 Introduction

Most of the research with distributed discrete event simulation and the conservative time synchronization algorithms has revolved around network queuing models. In terms of a network model RIZSIM can be thought of as a fully connected network where any object can impact or communicate with any other object. Also, objects can be dynamically created and destroyed. The service time or delay time at each node is zero. Given the constraints of this worst case situation, parallel speedup, if achieved, will provide valuable insight into parallelization of discrete event simulations in general.

4.2 Computational Complexity

The bulk of the floating point calculations in this simulation are in the sensor_check function. A sensor check calculation between two objects consists of approximately 40 floating point calculations. Remember that a single call to the sensor_check function results in sensor check calculations between the primary sensing object and every other object in the simulation. Therefore the order of complexity for a single event execution is order 'N' where 'N' is the number of players in the scenario. Although the number of floating point calculations between two objects in the sensor check is relatively insignificant, the total number of floating point calculations is directly proportional to the total number of players in the scenario. In a single processor execution of this simulation as the number of players increases the floating point computational complexity increases in terms of number of floating point calculations executed.

The fact that the inherent computational complexity is small and increases as the number of players increases indicates that, for an attempt to achieve parallelization speedup, some sort of data or player partitioning would be appropriate.

4.3 Data Partitioning

The traditional approach to data partitioning in a battle simulation has been to partition the battlefield. In fact many battle simulations or war games start out with a partitioned battlefield. With a partitioned sequential simulation the first logical step in parallelization is to put each sector or group of sectors on separate processors. In that case, each processor would be responsible for keeping track of the battle objects that are present in its sector or sectors. Normally in this type of simulation there are rules for movement between sectors and rules for sensing or seeing across sector boundaries.

The premise for speedup of the partitioned simulation on a parallel system is that battle objects physically separated by such a distance that they 'reside' on separate processors would normally not perform any function or operation that could impact one another. Therefore, two objects on separate processors could execute events simultaneously and thus achieve simulation speedup.

Also, each processor is only looking at a subset of the total number of objects in the simulation, and therefore each event execution on a single processor should take less time to complete. Another way to look at this is that when an object executes an event on a logical process it is searching for the next object with which it will interact. When that object is found, an event is scheduled reflecting that future interaction. With objects distributed among logical processes in a partitioned simulation the search space for any given event execution becomes smaller. And thus the search time will decrease. This is the same as sorting data in order to facilitate searching through that data. In fact it is entirely possible for a partitioned battlefield simulation to run faster on a single processor than a non-partitioned battlefield simulation.

At first glance, it seems that battlefield partitioning is the natural approach to take, but this method becomes somewhat more complex when cross partition movement and sensing is taken into consideration.

Cross partition movement presents the possibility of rollback in the optimistic synchronization paradigm, and it could prevent two events on separate processors from executing simultaneously in the conservative paradigm.

As an example demonstrating this problem lets look at a colliding pucks simulation on a partitioned table. Although colliding pucks is less complex than a battle simulation, it presents many of the same problems for parallelization that a battle simulation presents (13:56). For this example a conservative time synchronization protocol is in use. The conservative method demands that only *safe* events be executed. That is to say, when an event is executed there is no possibility of an earlier event occurring.

In this example there are three processors: 'A', 'B', and 'C' see Figure 4.1 Each processor is responsible for event execution for the pucks on its sector of the table. Puck #1 will impact the lower bumper at time 10, so the time of the next event for puck #1 on processor 'A' is 10. Pucks #2, #3, and #4 are on processor 'B', are all traveling in the same direction and will impact the lower bumper at time 15. Puck #5 is resident on processor 'C' and is traveling from right to left and will cross the boundary at time 5. Puck #5 will subsequently strike puck #4 at time 6. This causes a chain reaction resulting in puck #2 crossing the boundary between 'A' and 'B' at time 8 and striking puck #1 at time 9.

Even though pucks #1 and #5 are at opposite ends of the table, puck #5 has indirectly impacted puck #1 resulting in a new next event time of 9 instead of 10. It appears that the only event that is safe to execute is the event at time 5 on processor 'C'.

For processor 'A' to be certain that it is safe to execute the event at time 10, processor 'A' must be able to lookahead from the current time to time 10 and be able to tell that no intermediate event will occur. One thought is to use application specific knowledge to help determine when events are safe to execute. For example, processor 'B' must be able to tell processor 'A' the earliest possible time that a

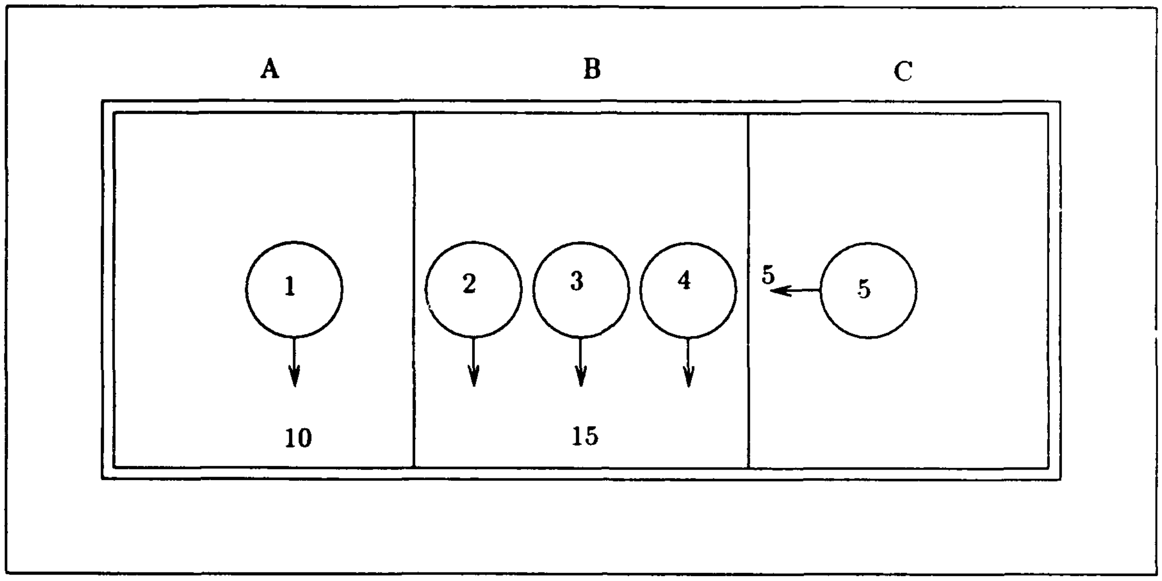

Figure 4.1. Conservative Colliding Pucks

puck might cross from 'B' to 'A'. Knowing the velocity of puck #2 and its location in relationship to the boundary with 'A', processor 'B' could provide the potential crossover time to processor 'A'. That would provide an earliest safe time for processor 'A' and processor 'A' could process events up to that time. This is analogous to Fujimoto's Lookahead discussed in Chapter 2.

The problem of predicting the earliest boundary crossing time for processor 'B' becomes much more difficult if one allows for the fact that puck #2 might increase its velocity based on a collision with a faster puck. Then not only does the time until crossover become shorter and the safe time for processor 'A' become smaller, but the overall prediction complexity increases. Processor 'B' must now determine which collision between pucks in its partition might result in an earliest boundary crossing. To make this determination processor 'B' must also take into account the fact that a puck or pucks from other processors may imminently move into its partition and thus impact its collision calculations. In this case it becomes difficult to determine the earliest crossover time. This is an example of a simulation application with poor lookahead capability.

A battlefield simulation is similar to the colliding pucks simulation in that both contain object movement and interaction. Also, it is not unrealistic to expect that a battlefield simulation might contain objects such as aircraft that can increase velocity. Thus the difficulty of predicting the earliest boundary crossing time also applies to the battlefield simulation.

The problem becomes even more complex when an object in one sector has a sensor range that extends into one or more other sectors and those other sectors are the responsibilities of other processors. An example would be an attempt to model the sensing capability of the Airborne Warning and Control System (AWACS) aircraft. It is entirely possible that the AWACS would have the capability to *see* into every sector of the battlefield. The processor on which the AWACS is resident potentially must have access to the location information for every aircraft in the simulation. If that information is divided among the processors, some method of providing that information to the AWACS must be devised.

Using a strictly conservative protocol with a feeble lookahead capability such as the example described above, even though the table is partitioned across three processors, at any given instant only one event can be processed safely. The processor with the smallest next event time is the only processor that is safe to execute and proceed with the simulation. This implies that each processor must have knowledge of the pending event times on all of the other processors in order to determine which next event is safe to execute.

The heart of the problem both in the colliding pucks simulation and in a battle simulation is that both have very little or no lookahead capability. As Fujimoto states (7:22-23), and as the colliding pucks example demonstrates, a simulation distributed among processors that uses a conservative time synchronization protocol must have good lookahead in order to achieve good performance. *A conservative* *distributed simulation with no lookahead essentially results in a sequential execution among processors.*

4.4 Identical Time Events

It is possible to have identical time events execute safely and concurrently on two separate logical processes and thus achieve some speedup. The problem here is that some simulation applications, RIZSIM in particular, demand a sequential execution of identical time events. The outcome of the simulation varies depending on the execution ordering of identical time events. Also, in RIZSIM, simulation time is kept using double floating point numbers and identical time events are rare. So even if identical time events were able to execute concurrently, that type of concurrency would be rarely exercised.

4.5 Partitioned Search Space

A battle simulation has feeble lookahead capability and the opportunity for concurrent processing among processors is virtually lost. The hope for speedup in a battle simulation, with a partitioned battlefield and a conservative time synchronization protocol, lies in the fact that each processor, executing sequentially, is executing an event against a subset of the the total number of battle objects. The battle objects are partitioned among processors based on each one's location on the battlefield. This effectively partitions the search space among processors, and therefore each processor has a smaller search space. If one considers the fact that the search space can also be partitioned or sorted on a single processor execution of the simulation, the question remaining is whether or not this 'smaller search space' opportunity for speedup on multiple processors is enough to offset the overhead of lookahead calculations, boundary area collision and sensing, and object migration between processors.

4.6 RIZSIM, and Lookahead

There are several factors which contribute to the poor lookahead capability of the AFIT Battle Simulation (RIZSIM):

- Movement - The movement of objects in space is similar to that of the colliding pucks simulation. Battle objects may cross from one sector or processor to another at crossover times that are difficult to predict without first executing many calculations.
- Object Interaction - Just as pucks collide resulting in changed velocity vectors for each ball, battle objects may sense each other, evade one another, or create and launch missiles.
- Dynamic Object Creation and Destruction - Battle objects can be dynamically created and destroyed as the simulation progresses.
- Velocity Increase - Although in this simulation individual objects do not increase or decrease their velocities, an object can create and launch a missile towards another object and the missile normally has a velocity greater than the object that launched it.
- Identical Time Events - Testing indicates that, for this simulation, identical time events must be executed in the order in which they are inserted into the next event list. They must be executed in first-in first-out order. A different scheme such as last-in first-out for identical time events will result in different results for the simulation.
- Zero Time Increment Event Scheduling - A current event execution can result in another event creation and scheduling with a timestamp that is the same as the current event. This new, zero time increment event, is produced as a result of the sensor_check function and cannot be predicted prior to execution of sensor_check. The zero time increment attribute eliminates the traditional

Chandy-Misra conservative null message passing protocol as a potential protocol for parallelizing this simulation since their paradigm depends on the use of non-zero time increments in order to prevent deadlock.

4.7 RIZSIM and the Case Against Battlefield Partitioning

In RIZSIM there are no sectors or battlefield partitions. After analyzing some examples such as the colliding pucks simulation, and looking at the characteristics listed above, it is clear that a battlefield partitioning of RIZSIM would add little if anything to its lookahead capability. Also, battlefield partitioning does add significant complexity to the simulation in the form of cross boundary sensing and movement, and coordination among processors. This added complexity results in additional computation and message passing both of which cut into the potential speedup expected by partitioning the data.

4.8 RIZSIM and a Hybrid Approach

4.8.1 **Logical Process Definition.** In parallelizing RIZSIM with the hybrid approach each physical processor plays host to a single logical process. Each logical process contains an event list and a data structure that holds the current state of the entire simulation. That is to say the data structure holds every object's state information. This state information is the same as the observable parameters discussed in Chapter 1, and consists of variables reflecting information such as speed and current location. Although in each logical process only a subset of the total number of objects in the simulation are eligible to be in the event list, the logical process has access to every object's state information. *The single logical process per physical processor scheme simplifies implementation, and eliminates the overhead associated with multiple communicating logical processes on a single processor.*

4.8.2 **Time Synchronization, State Change Updates and Local Lookahead.** *Conditional Event Execution* - It has already been pointed out that without lookahead the conservative time synchronization paradigm is reduced to sequential execution among logical processes. Assuming that is the case, following every event execution, logical processes must transmit globally time of next event information. In this way every logical process in the simulation can determine which logical process has the minimum pending or next event time. Only the logical process with the smallest next event time is safe to execute. This paradigm fits in with Chandy and Sherman's (5:94-96) thoughts on conditional and definite events, and global determination of time of next event.

Event List Partitioning and Data Replication - In this paradigm instead of battlefield partitioning, each battle object in the simulation can be assigned to any logical process without regard to geographic location. Each logical process has a single next event list which contains future events for the battle objects that have been assigned to it. Also, each logical process maintains current up-to-date state information on every battle object in the simulation. During an event execution the state of the object associated with the event is updated locally, and at the conclusion of the event execution, state changes for that object are sent via a message to all other logical processes. Now instead of just global sending of next event time information, state change information is also transmitted.

The types of state changes that can occur as a result of an event execution must be determined prior to parallel implementation based on knowledge of the simulation application. An alternative is to send the entire state of the object whose state changed. This does not require application knowledge but results in larger message sizes.

Local Lookahead and Parallel Speedup - Up to this point data replication, event list partitioning, and conditional event execution will not only result in a sequential event execution across logical processes, but due to the message passing overhead will also result in simulation slow down. Speedup requires the application of optimistic methods. Dickens and Reynolds (6:161-163) propose a method of local rollback, whereby pending conditional events are optimistically executed but the results of that execution are not transmitted to other logical processes. Although Reynolds applied this concept to a message initiating model (queuing network), the same concepts can be applied to a self-initiating model (battle simulation).

In the battle simulation, when one logical process is executing safely the others can be precomputing their next event. The results are maintained locally, and once the pending conditional event becomes safe to execute, the results can be made *official* and transmitted globally in the update message. In this scheme each processor that is not safe to execute looks ahead one event into the future and essentially has a "leg up" on event processing when it does become safe to execute. Each processor takes advantage of its idle time by predetermining as much as possible the results of the next event execution.

This then is a hybrid approach that combines the idea of conservative conditional event execution with optimistic local lookahead to achieve speedup. The following example (fig 4.2) with two logical processes illustrates this scheme. Each logical process has access to global state information, and each logical process has knowledge of the other logical process's next event time. Logical process #1 has the minimum next event time so it can execute its next event safely. Logical process #2 realizes that it must wait, but instead of waiting idle, logical process #2 precomputes its next event and stores that information. Upon completion of its event execution logical process #1 sends updated state information together with a new next event time.

Upon receipt of the update information from logical process #1, logical process #2 determines that it is now safe to execute. The updated state information can impact the local lookahead of logical process #2 in three ways: Logical process #2 may have to do a complete recomputation, a partial recomputation, or no recomputation. With no recomputation or a partial recomputation, the previously computed results can be quickly updated and made permanent, and state update information

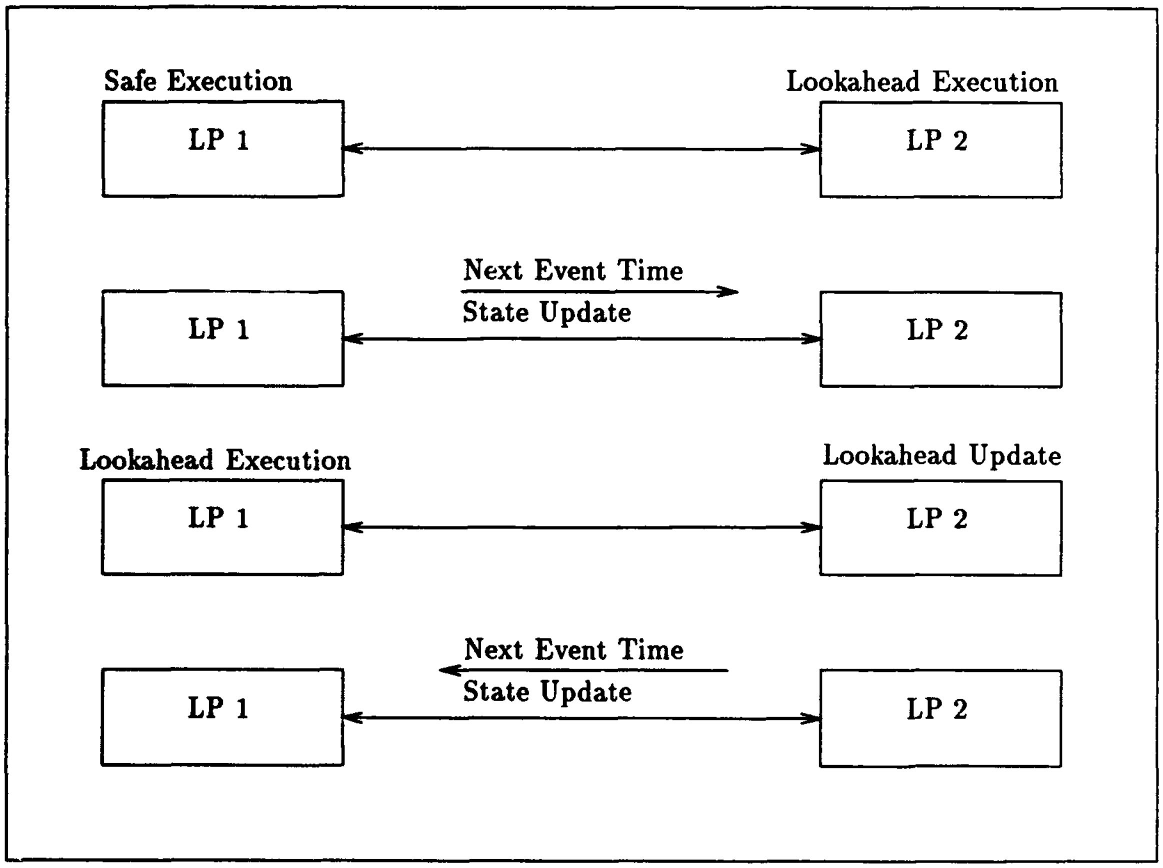

Figure 4.2. Hybrid Approach - Example

is sent back to logical process #1. In a best case scenario, assuming a favorable ratio of communications time to computation time and good interleaving of events between logical process #1 and logical process #2, linear speedup is approachable. If a logical process has completed a local lookahead and is not safe to execute, then it waits idle and processes received update messages until it does become safe.

4.9 Differences Between the Hybrid Approach and Battlefield Partitioning

In the hybrid approach data is replicated across all logical processes. That is to say every logical process has access to data reflecting the state of the entire simulation. In battlefield partitioning each logical process is only interested in, and only has access to information on objects in its sector and perhaps in its boundary areas. So for an event execution only a subset of the entire simulation state data is needed for computation. This is where battlefield partitioning achieves its speedup. The hybrid approach uses optimistic event execution to achieve speedup.

4.10 The Hybrid Approach and Speedup

The overall reduction in processing time in this approach is determined by what percentage of each event was precomputed using a lookahead while some other event was executing safely. The following three statements assume negligible communications and update processing overhead.

- If a logical process completes an event lookahead computation while some other logical process or combination of logical processes were executing safely then the effective execution time for that event is zero.
- If a logical process becomes safe to execute while in the midst of a lookahead computation of an event, then only the part of the computation that was completed prior to the logical process becoming safe represents processing time reduction.

- If an event is rolled back or if consecutive events are executed on one processor then there is no time reduction associated with those events.

Communications Overhead - Following each event, the logical process that just completed safe execution sends a broadcast update message to all other logical processes. This message is fixed size. Therefore for a simulation with a given number of events, as the single event computation time increases the communications overhead remains the same. Computation time can be increased by increasing the number of players or objects in the simulation. After a certain point the computation time becomes much greater than the communication time for each event, and therefore communication time will represent a negligible part of the simulation.

Update Processing Overhead - After a logical process becomes safe to execute and before it can make its lookahead results permanent it must execute a lookahead update. This update is needed because the previous event execution by some other logical process has changed the state (In the case of RIZSIM) of at least one and possibly two objects in the simulation. Using RIZSIM as an example, a complete event execution (either normal or lookahead) consists of a sensor check between the object associated with the event and every other object in the simulation. An update on the other hand will only consist of a sensor check between the object associated with the pending event and one or two other objects associated with the just completed event. As with communications overhead, the update processing time remains constant even if the number of objects in the simulation increases. Therefore a point can be reached where there are enough objects in the simulation so that the update processing time becomes negligible.

Assuming negligible communications and update time, total execution time for a simulation using the hybrid paradigm is defined by the following formula:

$$Hybrid\_Execution\_Time = (\sum_{i=1}^{n} P_i E_i) + (\sum_{d=1}^{m} P_d E_d)$$

where:

n = # of events in the simulation

i = event #

P = percentage of event execution during safe time

E = execution time for this event

m = # of events where there was a lookahead computation and the event was subsequently deleted

d = event # of an event on which a lookahead was accomplished and the event was subsequently deleted

This formula also assumes that the lookahead execution runs to completion prior to the logical process checking incoming update messages. Given that the execution time is constant for any given event the variables of interest are P and m.

Examining m first, an example from RIZSIM is in order. If one logical process conducts a lookahead execution and the object associated with that event is subsequently destroyed as the result of processing the update message then that event is deleted. The portion of the lookahead execution that was conducted while the logical process was safe to execute is wasted time and adds to the overall execution time. If the simulation is executing on a single processor there are no events of this type. With RIZSIM this occurs relatively infrequently, but for completeness this phenomenon must be taken into consideration.

The variable of most interest is the P on the left side of the plus sign. The following observations can be made about this variable:

- If the simulation is executed on a single processor then $P = 100$ for every event.
- If a logical process becomes safe to execute and must rollback and re-execute the next event then the value of P can range from 100 to 200. The value will be

closer to 100 if the logical process becomes safe to execute toward the end of its lookahead computation. The value will be closer to 200 if the logical process becomes safe to execute just after it has started its lookahead computations.

- If there is no rollback and the logical process becomes safe to execute after the lookahead computations are completely finished then the value of P is zero for this event.
- If there is no rollback and the lookahead computations are in progress when the logical process becomes safe to execute then the value of P will range somewhere between zero and 100.

Figure 4.3 shows two examples that illustrate this formula. The logical processes are on separate processors and are executing concurrently. The events are overlaid on a single time line in order to show what would be the single processor total execution time versus the multiple processor reduced execution time. The double lines between events serve to separate the events and indicate some overhead for communications and update message processing.

In example 1, for events #1 and #2 the values for P are between zero and 100. Each of those events became safe to execute while in the midst of the lookahead calculation. Therefore the lookahead execution time prior to the event becoming safe to execute represents 'saved time'. The portion of the event lookahead that was executed while the event is safe to execute is added into the overall execution time. The total execution time can be determined by summing the portion of each event that is executed in the safe area. The event execution time outside of the safe area represents time saved.

Consider the following values for E_i and P_i:

$E_1 = 100$

$E_2 = 100$

$E_3 = 100$

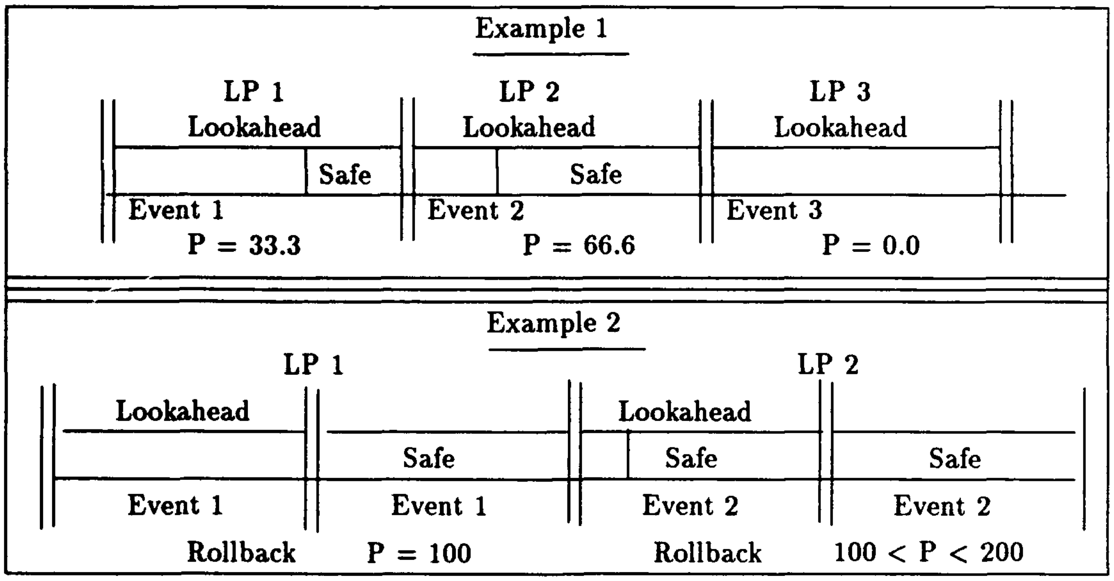

Figure 4.3. Hybrid Execution Time

$P_1 = 33.3$

$P_2 = 66.6$

$P_3 = 0.0$

$$HybridExecutionTime = 33.3\%(100) + 66.6\%(100) + 0.0(100) = 100$$

$$Speedup = \frac{300}{100} = 3$$

This portion of the simulation achieved linear speedup.

Example 2 shows the impact of rollback. Logical process #1 has completed a lookahead but must subsequently rollback and re-execute the event during its safe time. Since the lookahead was conducted during what normally would have been idle time there is no slowdown, but neither is there speedup. Logical process #2 became safe to execute immediately after beginning its lookahead. Following the lookahead, logical process #2 processed its update message and determined that it was now safe to execute and that a rollback was needed. Therefore event #2 was re-executed. In this case there is simulation slowdown since one event is essentially executed twice.

The underlying premise here is that rollbacks such as the one in example #2, event #2 will occur relatively infrequently. Intuitively one would think that this would be the case. As the number of processors and the number of objects in the simulation increase the chances that an event execution on one processor impacts the immediately succeeding event on another processor decreases. Test results (Chapter 6) bear out this premise.

The formula for simulation speedup is as follows:

$$Speedup = \frac{Single\_Processor\_Execution\_Time}{Hybrid\_Execution\_Time}$$

The lower the value of P for each event in the simulation, the lower the Hybrid Execution Time and the better the parallel speedup. An interesting aspect of the Hybrid formula lies in the fact that for a given set of P_i as the computational intensity increases the the amount of speedup remains the same since the Hybrid Execution Time will increase proportionately with the Single Processor Execution Time. This observation of course assumes negligible communications and update processing overhead. The challenge of lowering P in a parallel hybrid simulation is discussed in the next section.

Measuring 'P' - Without interrupt servicing of incoming messages it is difficult to measure the exact value of P for each event execution. The logical process cannot measure the exact instant in its lookahead calculations that it received an update message with contents indicating that this logical process is now safe to execute. However it is possible to measure the number of events where the value of P will lie somewhere between zero and 100. These are the events where a lookahead has been completed and the results are made permanent, perhaps after a quick update. In other words the results of the lookahead are actually used and no rollback of any type was executed. As the value of P approaches zero for each event the multiple processor execution comes closer to linear speedup. The following Use Lookahead Ratio reflects the percentage of the time that a beneficial lookahead was accomplished:

$$Use\_Lookahead\_Ratio = \frac{Used\_Lookaheads}{Total\_Number\_of\_Events\_in\_the\_Simulation}$$

4.11 The Hybrid Approach and Factors Impacting Speedup

There are two factors impacting speedup in this approach: event interleaving and the ratio of computation time to message passing time. Computation to communications ratio plays a role in every parallel application, but the factor unique to this paradigm is event interleaving.

4.11.1 **Event Interleaving.** Event interleaving refers to the degree to which event executions are interleaved among logical processes. Since each logical process only looks ahead one event, consecutive event executions within a single logical process result in loss of speedup. The following trivial example demonstrates this. Logical process #1 in Figure 4.4 is executing safely, and at the same time logical process #2 is doing a lookahead computation. When #1 finishes, it sends an update to #2. Logical process #2 has precomputed results so it merely updates the results based on the information received from #1 and sends out its update message. If we assume negligible message passing time then two events have been executed in essentially the time it takes to execute one event.

In Figure 4.5 logical process #1 has two consecutive safe events to execute before process #2 is safe to execute. In Figure 4.4 the first two events are executed concurrently in almost the same time as one event, and the the second figure the first two events are executed sequentially on processor #1 and therefore speedup is lost.

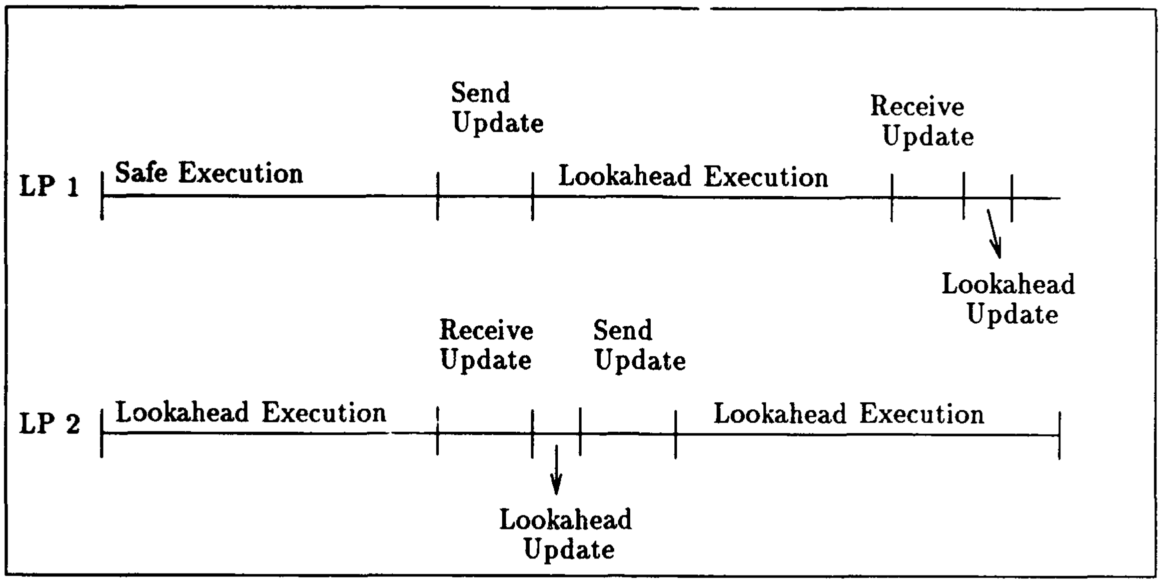

Figure 4.4. Event Interleaving

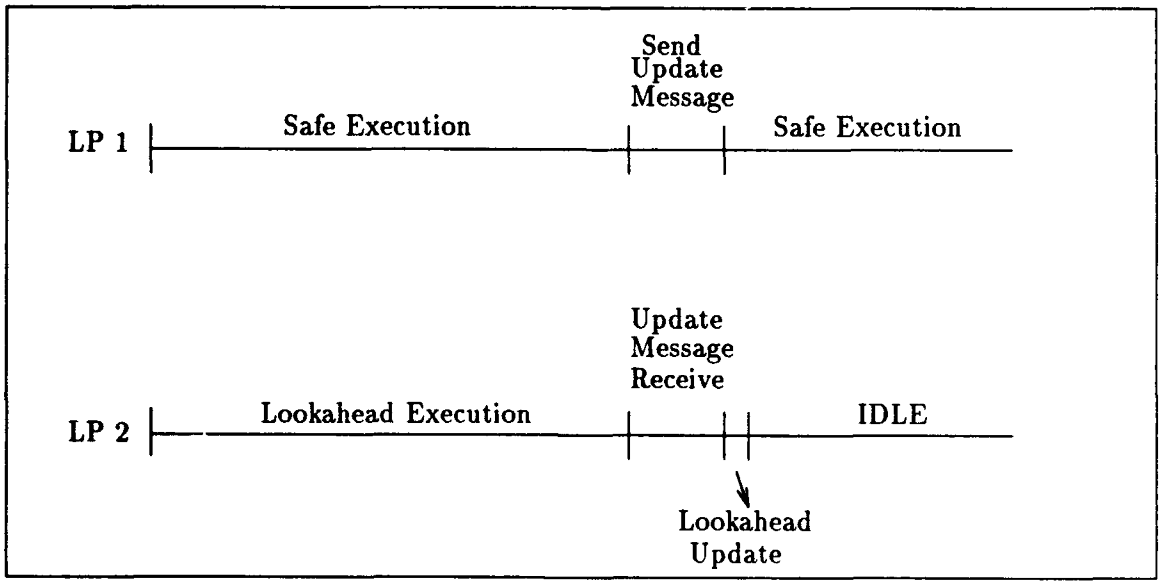

Figure 4.5. Consecutive Events Hurt Speedup

4.12 Hybrid and Load Balancing

Although this initial research does not delve into load balancing, intuitively it seems that load balancing with the hybrid approach is fairly simple. The goal of load balancing in the hybrid approach is to ensure good event interleaving. Since every logical process has access to the next event time of every other logical process, the logical process that is currently safe to execute can lookahead into its event list and determine with a good degree of certainty whether or not it will be executing consecutive events based on the timestamps of those events and its knowledge of other logical process's next event times. If it will be executing consecutive events then load balancing can be enacted by sending one or more events to other logical processes for insertion into their event lists. Further research would be needed to determine if this approach would be fruitful.

4.13 Hybrid and Linear Speedup

Let's consider a perfectly interleaved simulation with eight logical processes and therefore eight processors. There is one aircraft on each logical process and each aircraft executes five events. There are forty events in the complete simulation. A single event execution takes 7 time units. In this example initially update processing time is taken into consideration. An update execution based on the incoming update message from one logical process takes 1 time unit. This model ignores communication overhead.

Figure 4.6 shows the execution timeline for this simulation. While logical process #1 is executing its first event the other logical processes are doing their lookahead calculations. While logical process #1 is doing its lookahead calculation the other logical processes are updating their lookahead calculations and broadcasting the results. An update execution takes one time unit therefore updates from seven other logical processes takes seven time units. Note that logical process #1 executed the first safe event in the simulation and therefore it has no update time immediately following its initial event execution. Logical process #2 executed the second safe event in the simulation and needed one time unit to process the update from logical process #1 following its lookahead computation. Logical process #3 will need two update time units and so on. Logical process #8 needs seven time units to process the seven updates from the seven other logical processes. Following the second and subsequent lookahead executions each logical process will need to process seven updates, one from each of the seven other logical processes. Logical process #8 turns out to be the "long pole in the tent" and summing the time values along logical process #8's timeline results in a total execution time of 70 time units. This is the total execution time for the perfectly interleaved simulation on eight processors.

A single processor execution of this simulation results in a execution time of 280 time units (40 * 7). Speedup for the eight processor perfectly interleaved model is 4.0 (280 / 70). By reducing the update time speedup increases and as the update time goes to zero speedup approaches eight (linear speedup).

It not unrealistic to expect update times to be much smaller than normal execution and lookahead times. A normal execution and a lookahead involves sensor checking all of the aircraft in the simulation, whereas an update only sensor checks the aircraft whose state data has changed based on the incoming update message. So in this case if there were 512 aircraft in the simulation then a normal or a lookahead execution would take 512 time units and a complete update would take 7 time units.

Although it is easy to realize small update times in a realistic simulation, the real challenge is to realize perfect interleaving in a realistic simulation.

4.14 RIZSIM and Hybrid - A High Level Example

Consider a RIZSIM scenario with 50 Red Aircraft against 50 Blue Aircraft running on two processors.

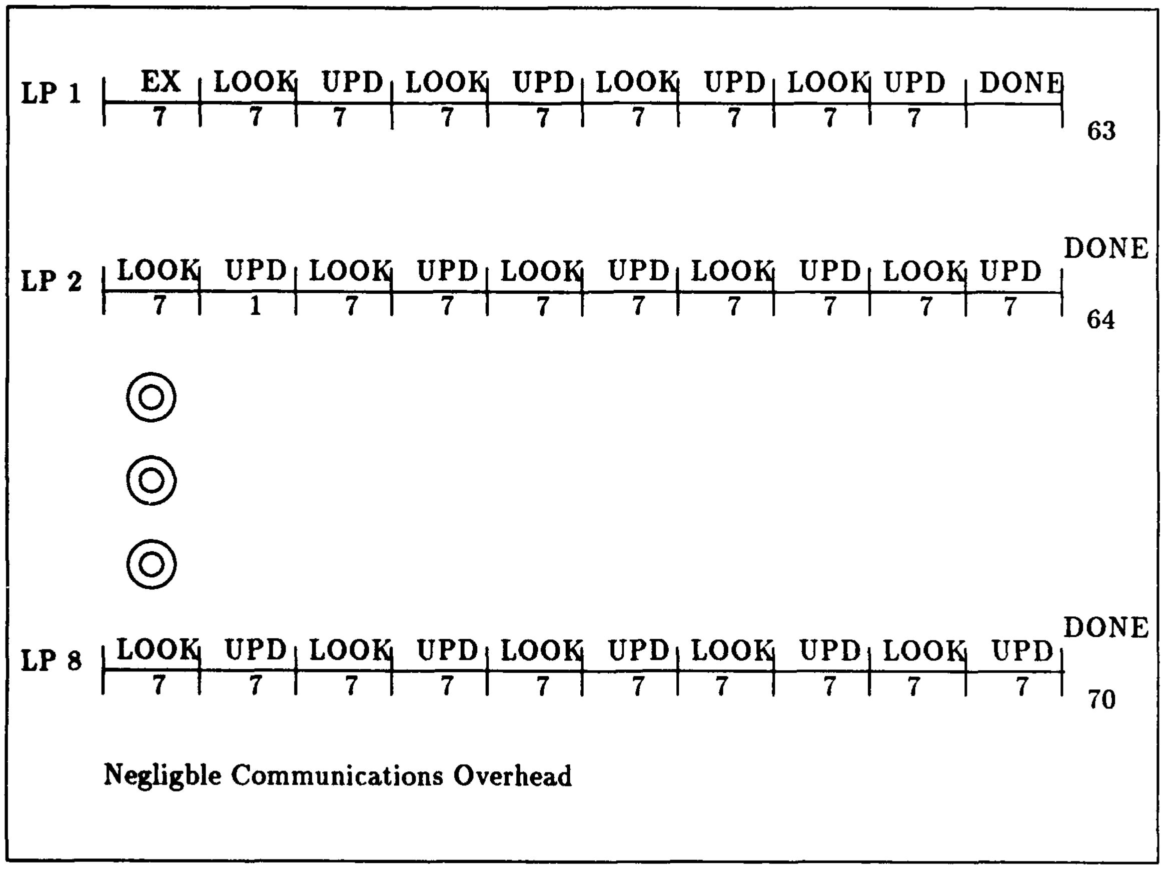

Figure 4.6. Logical Process Timeline

- **Scenario** - The scenario file consists of 100 lines of data. Each line describes the initial characteristics or state of each aircraft. Each line also contains the battle plan for each aircraft, in the form of a list of routepoints and a list of target types. The two logical processes, logical process #1 and logical process #2, both read in the entire scenario. Therefore each logical process has access to the complete set of global state information for this simulation run. After each logical process has read in the scenario, initial events are scheduled.

- **Initial Event Scheduling** - This is the point where the event list gets partitioned. In this example, a round robin type of initial event scheduling is used. As it turns out logical process #1 gets all of the even number aircraft and logical process #2 gets all of the odd number aircraft. There is an even mix of red and blue aircraft on each logical process. Even though each logical process has access to global state information on every aircraft, each logical process will only execute events for the aircraft that it has initially scheduled. Aircraft do not migrate from one logical process to another as in the battlefield partitioning paradigm.

- **Initial Event Execution - Tiebreaking** - In this case there are 100 initial events, 50 on each processor, all of them with an event time of 0.0. With this conservative approach only the event with the minimum safe time is safe to execute. In RIZSIM, the order in which identical time events are executed matters. That is to say that different execution ordering of identical time events leads to different results. Therefore a tiebreaker is needed for identical time events. The details of the tiebreaker algorithm are described in Chapter 5. Basically it consists of associating a global event number with the event at the time it is inserted into the next event list. If there are two logical processes with two pending events with identical timestamps then the pending event with the lowest event number executes first.

- Lookahead - While logical process #1 was executing the event for Aircraft #1, processor #2 was doing a lookahead event execution for the next pending event in its list. The pending event on processor #2 is in this case associated with Aircraft #2. After receiving the updated information from processor #1, processor #2 updates its lookahead calculations, makes its lookahead calculations permanent, and sends an update message to processor #1.
- Missile Creation - When an Aircraft launches a missile, a missile object is created on that Aircraft's processor. An Ordnance Released event is scheduled on the processor, and the outgoing update message contains information relevant to the new missile so that all other processors can update their global state information by adding the new missile to the simulation.
- Ordnance Reached Target - An ordnance reached target event results in the destruction of the missile and its target. This event may result in the deletion and rescheduling of future events. For example if some other aircraft has scheduled a made sensor contact event with an aircraft that gets destroyed then that made sensor contact event must be rescheduled since the aircraft has been destroyed and no longer exists. The update message contains the object identification numbers of the missile and the aircraft that were destroyed. This type of update message can result in local rollback or recomputation of the lookahead ii the pending event or the projected event are related to either of the two objects that were destroyed.

The simulation proceeds by continually executing safe events and precomputing conditional events until the event list on every processor is empty. At that point the simulation terminates.

4.14.1 **Update Message Contents.** Following each safe event execution the logical process sends out an update message to all other logical processes. The contents of this message are as follows:

- Time of Next Event - The sending logical process inserts the time of its pending event into the message. This gives all logical processes the capability to determine which logical process will safely execute next.
- Update Information - This information is application specific. In RIZSIM this information consists of location, velocity, and routepoint changes. Since all logical processes start out with the same data on all aircraft in the simulation, only change information need be sent in this message. If a missile is created as a result of the event execution then information relating to the missile's initial state is included in the message.
- Tiebreaker Information - This information is application specific. As mentioned previously, in order to achieve consistent results RIZSIM demands that identical time events be executed in a certain sequence. A globally unique tiebreaker number is associated with each event and must be included in the update message. Two event numbers are sent in the simulation. One is the event number associated with the pending event. The other is the next available event number for use in scheduling future events.

4.15 Summary

Each physical processor contains a single logical process. Each logical process contains a subset of battle objects. A logical process's subset of battle objects are 'native' to that logical process and can have events in that process's event list. A logical process has global knowledge of state information of every battle object in the simulation. Only the pending event with the global minimum next event time is safe to execute. Following execution of a safe event an update message is sent to other processors. This message contains the changes to the state information impacted by execution of the safe event. The message also contains the new time of next event. In this way global state information remains consistent and minimum next event time can be determined.

In order to realize speedup, while the safe event is being calculated, the other non-safe logical processes are precomputing their next events. Once a previously non-safe processor becomes safe to execute, the precomputed event information can quickly be made permanent.

Speedup is dependent on the amount of event execution interleaving among logical processes. The greater the amount of interleaving the better the chances for speedup. Speedup is also dependent on the computation to communication time ratio. The lower the communication time and the greater the computation time the greater the opportunity for speedup.

V. The Hybrid and RIZSIM - Top Level Design

5.1 Design Considerations

5.1.1 **Software Integration.** The original plan was to integrate three software packages: RIZSIM (the battle simulation), a locally developed simulation driver, and SPECTRUM (protocol design and testbed). The three packages are not completely compatible with one another. For instance, each package contains a simulation driver, only one of which can be used. Also, all three have defined data structures with the same name and different fields. For these reasons it was not possible to completely integrate all aspects of each package.

5.1.2 **Simulation Integrity.** The measure of success for simulation integrity is that the simulation must produce the same output from two, four, or eight processors, as it does on one processor. The graphical output file provides a detailed trace of what occurred during the simulation run. The integrity of a multiprocessor simulation run can be determined by comparing the graphical output file with that of a single processor run. A given input scenario always produces the same output, therefore for a given input scenario the results must always be the same whether the simulation is executing on one processor or eight processors.

5.1.3 **Multi-Processor Scaling** The design should support easy running on various numbers of processors in order to facilitate testing. SPECTRUM provides facilities for easily executing a simulation on various size 'cubes'.

5.2 Parallel Design

The design consists of four parts. The first part relates to the initialization, the second to the conservative distributed synchronization protocol, the third to the optimistic lookahead and rollback, and the last to simulation termination. Before describing each of those parts it is worthwhile to describe the major data structures associated with this system.

Data Structures

- Event List - Each logical process has a time ordered list of events to be executed. Each logical process has its own unique event list. By unique I mean that a given aircraft will have events in only one event list associated with one logical process. The initialization design (below) describes how aircraft are partitioned among logical process's event lists.
- Object Data Structure - This structure consists of the information relating to the state of the simulation. It contains all of the aircraft, each one uniquely identified by a number, and location, velocity, sensor range, plan of movement, and list of targets for each aircraft. In this implementation this structure consists of an array of pointers to the aircraft attributes. For instance array element number five is a pointer to the record that contains all of the state information associated with aircraft number five. This data structure is replicated across all logical processes and must remain consistent across all logical processes. During initialization each logical process starts with an identical copy of this data and as a logical process changes the data values within the structure it sends a message to all other logical processes detailing the change. This data structure is part of the original sequential simulation. Assuming that all of the processors in the parallel configuration are identical, if a single processor can run the simulation then a multiple processor configuration will work.
- Next Event Time Array - This array contains the time of the pending or next event on each processor or logical process. For example array element number two contains the time of next event for the pending event in the event list in logical process number two. This array is replicated across all logical processes

and must be consistent across all logical processes. Each logical process has access to the pending event time of every other logical process. Therefore all logical processes can easily determine which logical process's pending event time is the minimum pending event time. With this knowledge the logical process with the minimum pending event time can safely execute.

The data structures described above provide the basis for design of this simulation with this conservative protocol. Characteristics of RIZSIM demand the use of two other data structures that must be globally replicated and consistent across all logical processes.

- Tiebreaker Array - As mentioned previously RIZSIM demands that identical time events be executed in a certain sequence in order to preserve simulation integrity. The tiebreaker array is an array of event identifiers associated with the pending event on each logical process. If two or more pending events have identical event times then the pending event with the smallest event identifier is the one that can [illegible] safely executed.
- Highest Object I[illegible]er - Since objects can be created dynamically in the form of launched [illegible]iles and these objects need unique number identifiers a global variable is maintained which contains the value of the next available identifier number.

5.2.1 **Initialization Design.** During initialization the Spectrum software loads an identical copy of the application software on each processor and sets up a logical process to physical processor mapping. This is the only use made of the Spectrum testbed in this parallel simulation. After each logical process is loaded onto its respective processor (one logical process per processor), the application software RIZSIM begins executing on each processor. The first step RIZSIM takes is to read in the input scenario. Each logical process reads in t[illegible] scenario from the scenario data file via the RIZSIM function "read_datafile". Remember that each line of the scenario data file contains complete information about one aircraft in the scenario including its initial location and velocity vectors together with its plan of movement, and list of target types. In this way every logical process in the simulation has a complete copy of the input scenario. During this part of initialization RIZSIM also initializes the graphical output data file. In order to ensure that duplicate information is not written to the output data file some changes were made to "read_datafile" which only allow one logical process to initialize the output file.

At this point in the simulation each logical process has a complete copy of the scenario and the graphical output data file has been initialized. Also, since new objects can be created in the form of missiles, a global number representing the next available aircraft identifier is maintained by each logical process. For example if initially there are 100 aircraft in the scenario then each logical process realizes that the next object identifier number will be 101 and the number 101 will be assigned to the first missile that is created.

5.2.2 **Initial Event Scheduling.** The next step is to schedule initial events for each logical process. This is accomplished in the RIZSIM function "schedule_init_events". In this step the aircraft get divided among the logical processes. Once an initial event is scheduled for an aircraft on a particular logical process then that aircraft becomes "native" to that logical process and all subsequent events for that aircraft will be with the same logical process. Since all logical processes have access to the data for every aircraft the initial events are scheduled based on unique aircraft identifier, physical processor node identifier associated with the logical process, and the number of logical processes in the simulation. The algorithm is as follows:

IF (aircraft identifier MODULUS number of nodes EQUALS node identifier) THEN schedule an initial event for this aircraft.

Processor #	Aircraft ID
0	4, 8, 12, 16
1	1, 5, 9, 13
2	2, 6, 10, 14
3	3, 7, 11, 15

Table 5.1. Aircraft to Processor Assignment

For example, in a 16 aircraft scenario with four processors or logical processes the aircraft will be divided among processors as in Table 5.1.

Because of characteristics of the input scenarios and the manner in which RIZSIM executes initial events, this method of initial event scheduling results in a simulation that is initially perfectly interleaved. If there are 256 aircraft in the simulation then the first 256 events in the simulation are perfectly interleaved.

One nice feature of this paradigm is that after initialization no more checks or efforts are needed to determine which aircraft are native to which logical processes. If an event is executing for aircraft number four the state data for aircraft number four is updated locally during the event execution. Following event execution the changes for the state data for aircraft number four are sent to all other logical processes. The other logical processes don't care where aircraft number four exists, all they need to know is that they must update their state information for aircraft number four.

5.2.3 **Tiebreaking.** Another aspect of initialization that also applies throughout the simulation run relates to tiebreaking when two events have identical timestamps. In RIZSIM the order of execution of two identical time events makes a difference in the outcome of the simulation. In order to preserve the correct sequence of execution of two identical time events in this parallel system, as in a sequential system, a tiebreaking mechanism is required.

In RIZSIM the tiebreaking rule is based on time of insertion into the event list. Between two events with identical times the event that was inserted into the event list first will be executed first. In order to ensure that tiebreaking works correctly a globally unique event identification number is assigned to each event when it is inserted into the event list. If there are two pending events with identical times then the event with the smaller event number is safe to execute. The initial event number assigned to each event corresponds to each object's identification number. Object or aircraft number one has an event number of one, two of two, and so on.

Taking an example where there are 16 aircraft in the simulation, aircraft #1 on processor #1 executes the first event in the simulation. If this event results in another event scheduled, then that event will have an event number of 17. Upon completion of its event execution, the updated state information on Aircraft #1, the next available event number (18), and the next event time on processor #1 (0.0), are sent to the other logical processes.

5.2.4 Conservative Distributed Synchronization Design

. The conservative protocol ensures that only the logical process with the smallest pending event time executes. The control structure for this paradigm follows:

```
WHILE NOT DONE
        process_incoming_messages
        IF safe_to_execute THEN
             do_event
        IF send_message_flag THEN
            send_outgoing_messages
END WHILE
```

5.2.4.1 Function Descriptions -

process_incoming_messages - the function process_incoming_messages performs several functions:

- Updates next event time for the sending logical process in the Next Event Time Array.
- Updates next event identifier number for the sending logical process in the Tiebreaker Array.
- If the sending logical process created a new missile object as a result of its last event execution then the receiving logical process uses data from the message to create the same missile. This ensures that the object state data structure remains consistent across logical processes. Also, if a missile was created then the highest object identifier variable must be incremented.
- State changes, such as location and velocity, are updated in the object state data structure based on information provided in the update message.

safe_to_execute - This function determines which logical process has the minimum next event time by looking through the Time Array. If two logical processes have the same minimum time then next event identifiers are checked and the logical process with the smaller identifier number is the logical process that is safe to execute. Since event identifiers are globally unique there is no chance of a tie between event identifiers.

do_event - Event execution remains essentially unchanged from the sequential version with two exceptions. Whenever RIZSIM sc[illegible]ules an event then the build_message function is called, and the set_event_id function is called. The build message function provides the content for the outgoing update message, and the set_event_id function adds the event identifier number to the event to facilitate tiebreaking. The build_message function also sets the send_message_flag to true. The build_message function writes its data to an output message buffer and must first check and perhaps wait for the previous message to be completely sent.

send_outgoing_messages - The function send_outgoing_messages simply sends the update message to all other logical processes. It uses a non-blocking send with a one-deep output message buffer. Since there is a danger of the message buffer being overwritten by the build_message function, the build_message function first checks to ensure that all messages have been sent before writing to the output message buffer.

5.2.4.2 **Update Messages.** The message sent from one logical processor to all others following an event execution is called an update message and contains the following information:

- Time of next event in the sending logical process's Event List.
- Event identifier number of the next event in the sending logical process's Event List.
- If a missile was created then initial state data related to the new missile is included in the message. A missile creation also indicates to all logical processes that the globally unique object identifier number has been incremented.
- Normally an event execution results in a change of state for one of the objects in the simulation, therefore an object identifier together with change information is included in the message.

5.2.4.3 **Conservative Time Synchronization Design Decisions.** The most difficult design decision was whether or not to send multiple functionally cohesive messages after each event execution or to send one message that contains various types of data. The application RIZSIM presents the possibility of a single event execution creating a missile and changing the state of the aircraft that launched the missile. It was decided to send a single message following each event execution. Although this scheme forces logical processes to process a multifaceted message, I estimated that processing a single multifaceted message would be easier to implement than processing several distinct but related messages following each event. Composing and decomposing the incoming and outgoing messages is somewhat more complex, but the number of messages in the system is kept to a minimum.

and message processing following an event execution is easy to keep track of. Following an event execution the executing logical process will send exactly one message to all other logical processes.

5.2.5 **Optimistic Lookahead and Local Rollback Design.** The design up to this point allows the simulation to execute on a parallel hardware architecture. Up to this point, while one logical process is executing safely all others are on hold until one of them has the minimum next event time. The next step is to design and implement the software that will allow speedup by making use of that idle wait time. There are three main areas of the Lookahead and Rollback software design each of which is described below:

look_ahead - The function look_ahead makes a copy of the next event in the Event List and executes that event. The results of the event execution are saved in temporary storage. In the case of the application RIZSIM the lookahead involves executing the sensor check function and saving the results.

lookahead_update - Following a look_ahead execution a message will arrive with updated state information for one of the objects in the simulation. lookahead_update will execute, and based on the information contained in the message, lookahead_update will either update the results in temporary storage, leave those results alone, or throw them out completely. In the case of RIZSIM a lookahead_update normally consists of executing the function single_sensor_check. The function single_sensor_check was derived directly from the function sensor_check. The difference, implied by its name, is that single_sensor_check only runs a sensor comparison between the primary sensing object and the object whose state just changed, as reflected in the incoming update message.

make_lookahead_real - The function make_lookahead_real takes the data located in temporary storage following a look_ahead execution and makes that data permanent by transferring that data into the global data structures and scheduling appropriate events in the Event List.

The following is the simulation control structure program design language for incorporating both conservative time synchronization and optimistic lookahead:

```
lookahead_complete and use_lookahead = FALSE
WHILE NOT DONE
        process_incoming_messages calls lookahead_update
        IF safe_to_execute THEN
           IF lookahead_complete AND use_lookahead THEN
              make_lookahead_real
              lookahead_complete and use_lookahead = FALSE
           ELSE
              do_event
              lookahead_complete and use_lookahead = FALSE
        ELSE
           IF NOT lookahead_complete THEN
              use_lookahead = TRUE
              look_ahead
              lookahead_complete = TRUE
        IF send_messages THEN
           send_outgoing_messages
END WHILE
```

5.2.6 **Simulation Termination.** Simulation termination is coordinated through the host processor. For RIZSIM when a logical process's Event List is empty then that logical process is finished with the simulation. The logical process sends a simulation_complete message to the host. When the host has received a complete message from all logical processes the simulation is finished. When a logical process has an empty Event List the last update message contains a next event time of infinity and knowing that information all other logical processes refrain from sending further messages to the completed logical process.

5.3 Required Application Knowledge

One of the research questions deals in the area of application specific knowledge. Particularly, how much application specific knowledge does the parallel programmer need in order to parallelize a given application? The following summarizes required knowledge for parallelizing RIZSIM:

- Tiebreaking: Needed to know how identical time events are handled in sequential RIZSIM.
- State Updates - In order to "fill in" the contents of the update message, needed to know what state changes are possible as the result of a given event execution. Different events will result in different types of state changes. These relationships had to be identified and then a method of flagging those changes and providing the correct information to the outgoing update message had to be developed.
- Lookahead - For lookahead calculations, needed to know where state changes could be made. RIZSIM functions were copied and modified to make the state changes temporary for a lookahead execution instead of permanent as in a normal event execution.
- Lookahead update calculations - For lookahead update calculations needed to know all of the potential impacts of the state changes contained in an update message.
- Lookahead and Making the lookahead Permanent - For both Lookahead calculations and for making the lookahead calculations permanent had to know

the mechanics of a RIZSIM event execution so as not to disrupt the contents of the next event list or the state data structure.

5.4 Design Summary

As mentioned at the beginning of this chapter, one of the challenges was to try and integrate three different software packages: Spectrum, RIZSIM, and a Simulation Driver.

5.4.1 **Spectrum.** The goal of Spectrum is to provide a testbed that facilitates mixing and matching parallel protocols with various applications. The hope is that with a stable baseline test environment, better timing comparisons can be made. Spectrum provided a nice facility for defining logical processes and mapping them to processors. Spectrum also provided some of the basic functions needed to send messages between logical processes. Spectrum's logical process manager provides interfaces to the protocol specific functions called "filters". The examples provided were with queuing network simulations and it was not clear how the hybrid concept fit into Spectrum's concept of filters. The logical process manager functions provided with Spectrum were not used, instead the control structure described above was implemented.

5.4.2 **Simulation Driver.** The simulation driver built into RIZSIM was removed and replaced with the locally developed simulation driver. The locally developed simulation driver provides a suite of singly linked-list manager functions, which for the purposes of this research were replaced with doubly linked-list manager functions.

5.4.3 **RIZSIM.** The functions added as a result of parallelizing RIZSIM with this hybrid design fall into three areas: conservative time synchronization, global state change, and optimistic lookahead with local rollback. These functions are described above.

VI. Test Results

6.1 Simulation Integrity Testing

After each step in the software design and implementation process simulation integrity testing was conducted. This was to ensure that a simulation running on a single processor produced the same results as a simulation running on multiple processors.

Since the simulation produces a graphical output file that is a detailed tracc of aircraft and missile movement, creation, and destruction, that file was used to verify simulation integrity.

First a single processor run was conducted with scenarios containing the following aircraft numbers: 2, 4, 8, 16, 32, 64, 128, 256, and 512. Each of these runs produced a baseline graphical output file against which the multiple processor runs were compared.

For each of the scenario sizes simulation runs were conducted on two, four, and eight processors. The graphical output files were compared with the baseline, single processor, files using the UNIX 'sdiff' utility. This utility (using the '-s' switch) highlights the differences between two files.

The parallel implementation produced output identical to the single processor runs for the baseline suite of test scenarios.

6.1.1 Simulation Errors.

The baseline tests were conducted with aircraft speeds of five and missile speeds of 1000. Subsequent timing tests were conducted with missile speeds lowered to 20 units to make the scenario a little more realistic in terms of the relationship between missile and aircraft speeds.

At the missile speed of 20 the sequential RIZSIM "crashed" with scenarios of 512 aircraft on a single processor.

Also, with two of the 256 aircraft timing scenarios the total event counts were slightly smaller on eight processors vice two or four processors. Scenario 256b produced eight fewer events on eight processors out of 2674 events than on the single processor baseline run. Scenario 256e produced one less event on eight processors out of 2506 events than on the single processor baseline.

The reason for these errors both in the baseline sequential simulation and the parallel version are unknown. They may be related to the floating point calculation problem (See Appendix A.). In any case, the errors in scenarios 256b and 256d had no significant impact on the timing results for either run.

6.2 Timing Tests

In general any type of output during the simulation run can lead to unpredictable timing results since output flows through the 'host' process, and the 'host' process is subject to use from other users and the operating system. Therefore all simulation output during the timing runs was turned off.

6.2.1 **Timing Run Scenarios.** All of the test scenarios have the following characteristics:

- Aircraft only are used. No ground vehicles are used.
- There are equal numbers of red and blue aircraft.
- The sensor range for all of the blue aircraft is 15.
- The sensor range for all of the red aircraft is 10.
- Aircraft speed is 5 for both red and blue aircraft.
- The initial routepoint and the subsequent flight plan routepoints were randomly generated using the UNIX 'C' library random number generator.
- The number of routepoints initially given to each aircraft is 6.
- The missile speed is 20 for all missiles.

6.2.2 **Initial Event Interleaving** The characteristics of RIZSIM tiebreaking and the initial event round robin scheduling result in the initial set of events being perfectly interleaved for each simulation run. If there are 256 aircraft in the scenario then the first 256 events are perfectly interleaved. As a comparison several simulation runs were made with range partitioning of the initial events so that there was consecutive execution of the initial events on each logical process. Also, this round robin scheduling with the scenarios used for these tests resulted in an even mix of red and blue aircraft on each processor.

6.2.3 **Scenario Sizes.** Timing Runs were made with scenarios of size 64, 128, 256. Five random scenarios for each of these sizes was generated and run. Each scenario was run on a 'cube' size of one, two, four, and eight nodes.

6.2.4 **No-Missile Scenarios.** A missile launch results in a zero time increment event scheduling and execution. This causes two consecutive events to be executed on a single processor, and hence hurts event interleaving and cuts into the USE_LOOKAHEAD ratio. Therefore simulation runs were made with scenario sizes of 64, 128, 256, and 512 where no missiles were launched. This helped increase the amount of interleaving in order to determine the impact of event interleaving on speedup.

6.3 Timing Measurements

Each logical process is timed from the time it starts until the time it finishes. The timing mechanism is based on wall clock time using the 'mclock' function provided with the Hypercube 'C' library. Each logical process calculates its own execution time. The overall simulation execution time is based on the logical process with the longest execution time.

The speedup calculation is the time of execution on a single processor divided by the time of execution on multiple processors.

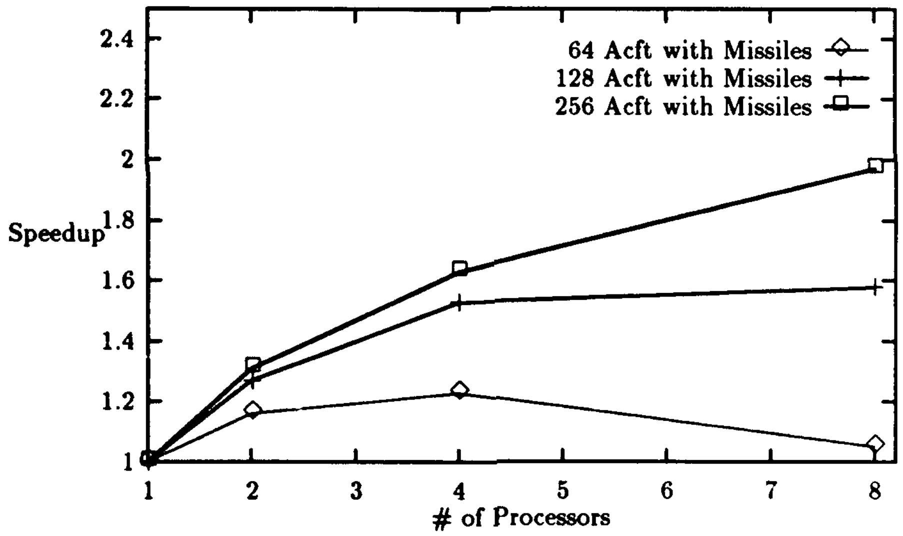

Figure 6.1. Simulation Speedup - with Missiles

Tables in Appendix B show the raw results for timing runs of various size simulations with and without missile launches.

The charts in figures 6.1 and 6.2 show the average speedup for various scenarios and various numbers of processors.

6.4 Findings

6.4.1 **Speedup/USE_LOOKAHEAD/Event Interleaving.** For every scenario as the number of processors increased, the USE_LOOKAHEAD ratio increased (indicating that interleaving increased) and the amount of speedup increased. Figure 6.3 shows the USE_LOOKAHEAD/speedup relationship for the 256 no-missile scenarios.

6.4.2 **Processor Assignment by Loyalty.** In order to assess the impact of the initial assignment of aircraft to processors, the initial event scheduling of the 256 aircrft scenarios were changed to get rid of the perfect event interleaving during

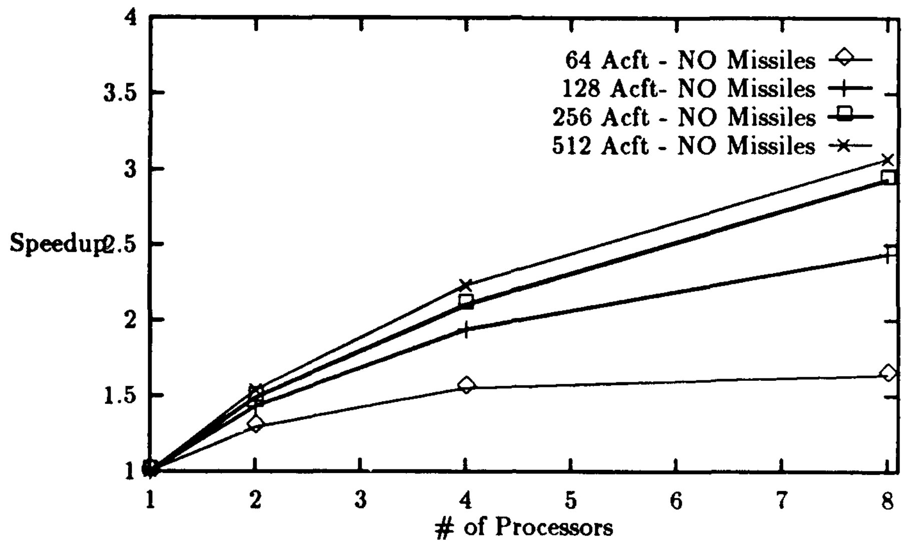

Figure 6.2. Simulation Speedup - without Missiles

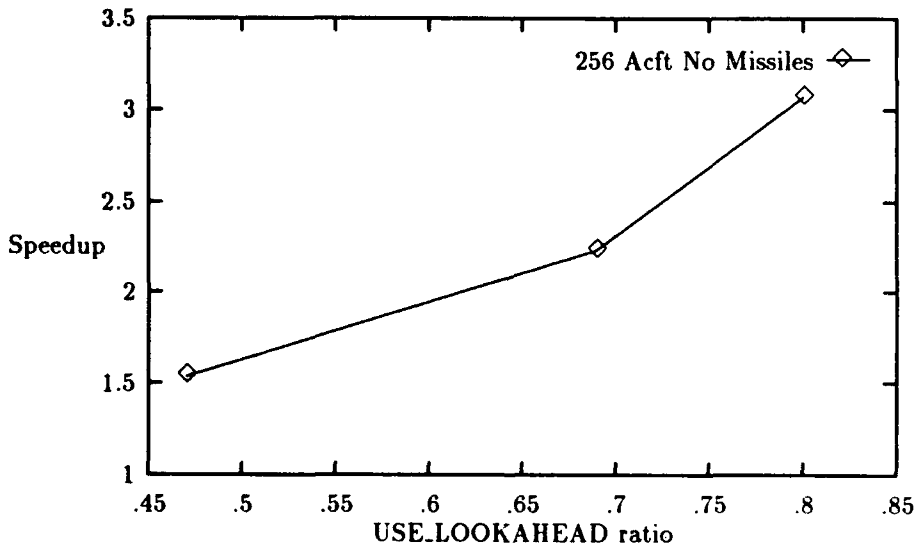

Figure 6.3. Speedup vs. USE_LOOKAHEAD

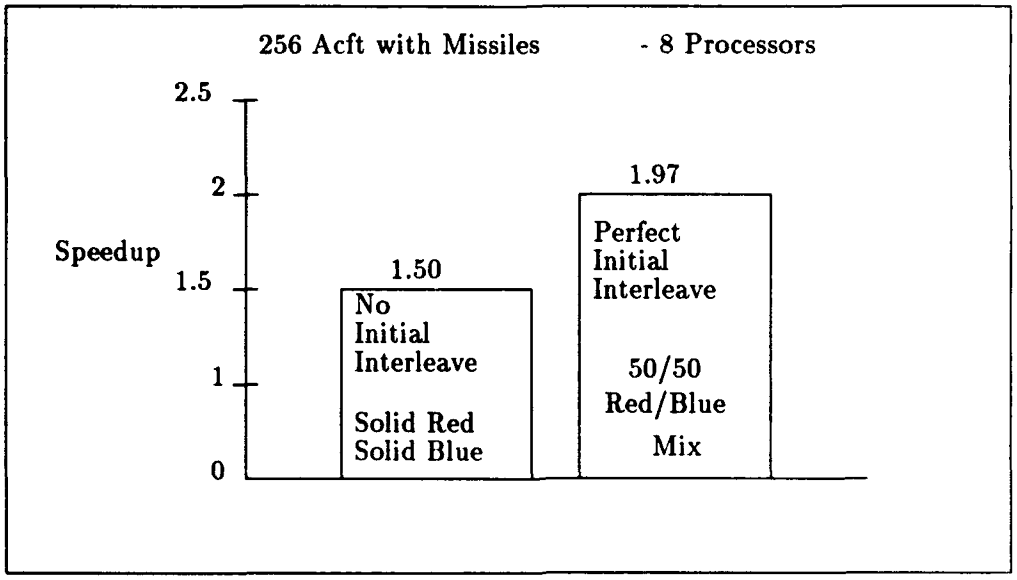

Figure 6.4. Initial Interleaved vs. Non_Interleaved (solid red-solid blue)

the initial event executions. The initial events still executed in the same order and the simulation results were identical, but instead of interleaving among processors, the initial events were executed consecutively on a single processor.

Figure 6.4 compares the 256 no-missile scenario with perfect initial interleaving (256 events) and an even mix of red and blue aircraft per processor to the 256 no-missile scenario with solid red on four processors and solid blue on the other four processors. This initial solid red/solid blue partitioning yielded eight interleaved events out of the first 256 events. Both initial partitionings produced identical results in terms of simulation integrity. The perfectly interleaved scenarios had a better USE_LOOKAHEAD ratio and better speedup. Analysis of the data reveals the the better USE_LOOKAHEAD ratio was due to almost entirely to the initial event interleaving bias.

Thinking that perhaps one of the keys to speedup is an even mixing of red and blue aircraft on each processor an initial scheduling was implemented that evenly

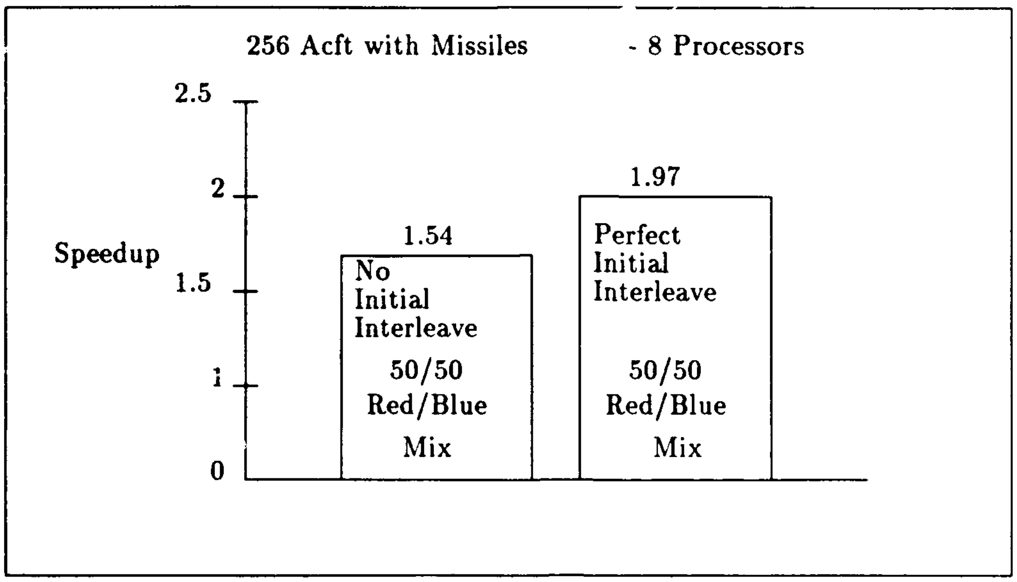

Figure 6.5. Initial Interleave vs. Non-Interleaved (even mix)

mixed red and blue aircraft on each processor and yet did not provide good event initial event interleaving. In this case there were 16 interleaved events out of the first 256 events. Figure 6.5 compares the non-interleaved even mix with the interleaved even mix. The speedup for the poorly interleaved solid red-blue versus the poorly interleaved red-blue mix (1.5 vs. 1.54) are virtually the same. This indicates that aircraft to logical process assignment by loyalty plays an insignificant role in speedup determination.

The same type of tests were run on the 256 aircraft (with missile) scenarios and similar results were obtained.

6.4.3 **Speedup Limits.** Although the 512 player scenario has many more event executions than any of the 256 player scenarios, the amount of speedup is the same (See Appendix B). This indicates that for a given number of processors and a given USE_LOOKAHEAD ratio the potential speedup has an upper limit no matter how much the size of the simulation increases in terms of number of players

Table 6.1. 256c Scenario, Spin 10, NO Missiles

# Processors	1	2	4	8	# Events
Scenario 256c	743.0	499.6	352.7	254.6	5984
Ave. Speedup	0	1.48	2.1	2.91	
Ave. Use_Lookahead	0	.47	.69	.80	

Table 6.2. 256c Scenario, Spin 100, NO Missiles

# Processors	1	2	4	8	# Events
Scenario 256c	2025	1374	965	696	5984
Ave. Speedup	0	1.47	2.09	2.90	
Ave. Use_lookahead	0	.47	.69	.80	

or computational intensity. To confirm this premise spinloops were inserted into the simulation. A spinloop artificially increases the computation time by executing a meaningless floating point calculation in a tight loop for a given number of iterations. Tables 6.1 and 6.2 show the results of these runs.

Note that although execution time increased due to the increased computation, the amount of speedup remained the same. After a certain point communication overhead no longer plays a primary role in limiting speedup. These results match the Hybrid_Execution_Time formula. As computation time increases, the USE_LOOKAHEAD ratio remains contstant and P for each event remains constant. The Hybrid Execution Time increases proportionately with the Single Processor Execution Time and therefore the amount of speedup remains constant.

6.4.4 **The Perfectly Interleaved Simulation** To get a better handle on the impact of perfect interleaving on speedup, and to verify the model on perfect interleaving presented in chapter 4, a scenario was developed that resulted in perfectly interleaved events among eight logical processes with no local rollbacks. The scenario consisted of eight aircraft split among eight logical processes with no missile launches, sensor contacts, or entered sensor ranges. The only events executed were the reached turnpoint events. Each aircraft had six reached turnpoint events. Spinloops were used to increase the computational intensity and thereby minimize inter-logical process communication impacts. A spinloop artificially increases the amount computation time in the scenario by executing meaningless floating point calculations in a tight loop.

The multiprocessor execution achieved a speedup of 3.98 when running on eight processors for this perfectly interleaved simulation. The model predicted a speedup of 4.0.

The perfectly interleaved simulation without the spinloops in the update module was also run and a speedup of 7.89 was realized. The model predicted a result of 8.0.

6.4.5 **Psuedo Realistic Scenario Execution.** In order to be sure that the observed speedup was not due to the initial random mixing of red and blue aircraft a scenario type was developed that has the following characteristics:

- The battlefield is evenly split into two areas, the red area and the blue area.
- Each Aircraft has six routepoints.
- All of the routepoints for the red aircraft lie within the red area.
- The blue aircraft initially start with two routepoints in the blue area. The next two routepoints are in the red area, and the final two routepoints for the blue aircraft are back in the blue area.
- Within each area for each aircraft the routepoints are randomly assigned.

These scenarios are realistic in the following sense: initially there is no contact between the red and the blue aircraft, the aircraft subsequently engage in battle as the blue aircraft penetrate the red area, and then the forces disengage as the blue aircraft return to the blue area.

The scenarios were run on eight processors and the results are shown in table 6.3. Note that the number of events increased slightly for the realistic scenario when compared to the original 256 Aircraft with missile scenarios in table 6.4. Also, speedup improved a little bit, as did the USE_LOOKAHEAD ratio.

Table 6.3. 256 Aircraft Scenarios - With Missiles - Psuedo Realistic

# Processors	1	8	# Events
Scenarios r256a	217.6	102.1	3294
r256b	211.1	99.9	3251
r256c	204.9	98.8	3181
Ave. Speedup	0	2.1	
Ave. Use_lookahead	0	.63	

Table 6.4. 256 Aircraft Scenarios - With Missiles

# Processors	1	2	4	8	# Events
Scenarios 256a	150.2	114.8	92.4	77.2	2538
256b	157.5	120.2	95.7	79.6	2674
256c	148.4	115.8	94.2	70.3	2551
256d	138.7	104.9	83.3	75.8	2318
256e	151.3	114.5	89.9	75.8	2506
Ave. Speedup	0	1.31	1.63	1.97	
Ave. Use_lookahead	0	.38	.52	.59	

Table 6.5. 256 Aircraft Scenarios - No Missiles - Various Routepoints

# of Processors	1	8	# Events	Ratio	Speedup
Scenarios 256 RP3	216.1	76.6	2444	.75	2.82
256 RP6	606.3	204.2	5980	.81	2.9
256 RP9	1139.0	371.3	10260	.82	3.0
256 RP12	1602.7	511.7	13740	.82	3.1

6.4.6 **Scenarios with Different Number of Routepoints.** Four scenarios one each of three, six, nine, and twelve routepoints for each aircraft were run on eight processors. Table 6.5 shows the results of those scenario runs. It appears that the amount of speedup levels off after a certain point as the number of events increases.

6.5 Summary

These results show that the Hybrid paradigm is effective at achieving simulation speedup. The best speedup ranged from two to three for eight processors depending on the type of scenario used and the number of events in the scenario. After a certain point the amount of speedup appeared to level off at approximately three for eight processors regardless of the increase in the number of events or the computational intensity.

The amount of event interleaving determines how much speedup will be achieved. As the number of processors increases the amount of interleaving increases and the amount of speedup increases.

The results remained consistent throughout all of the tests. No significant variations were observed by changing aircraft to processor mapping or by using a realistic type of simulation.

As demonstrated, perfect event interleaving can quite easily lead to significant speedup. The majority of the tests consisted of scenarios that resulted in random event interleaving. With the perfectly interleaved scenario speedup rose significantly and approached 8 for an eight processor simulation.

These tests indicate that the challenge is to develop methods to improve the amount of event interleaving in the hybrid paradigm.

VII. Summary, Conclusions, and Further Research

7.1 Summary

This paper described a method of parallelizing a battlefield discrete event simulation. This method combines elements of conservative time synchronization together with elements of optimistic computation and rollback.

Each hardware processor in this message passing architecture hosts one logical process. Each logical process maintains the state of the entire system. Although each logical process maintains complete state information, only a subset of the total number of objects in the simulation are resident in a given logical process's event list.

Of all the logical processes only the logical process with the minimum next event time can execute safely. During a safe event execution, the logical process updates its state information and schedules additional events for itself as appropriate. Following a safe event execution the executing logical process sends state update information and its new next event time to all other logical processes. With this globally distributed information all logical processes can determine which one of them is safe to execute next.

Up to this point the event list is divided among logical processes and at any given instant only one logical process is executing safely. In this way the events spread across multiple logical processes are executed in the same order as with a single processor simulation execution. This ensures simulation integrity. A multiple processor simulation execution produces the same results in the same order as the sequential version. This conservative paradigm by itself does not achieve speedup.

In order to achieve speedup an optimistic technique is implemented. While the one logical process with the minimum event time is executing safely, all other logical processes, instead of waiting idle, precompute as much as possible their next event. The results of the precomputation or lookahead, are kept in temporary storage. After receipt of the update message from the logical process that just finished executing safely, the other logical processes update their lookahead results. If one of these "lookahead" logical processes is now the safe logical process then it simply transfers the precomputed results in temporary storage into permanent storage, schedules new events, and sends out the update message.

7.2 Conclusions

This paper concludes by addressing the research questions posed in Chapter 1:

1. **Can a battle simulation achieve speedup using a parallel protocol hybrid consisting of conservative time synchronization and optimistic next event look-ahead on a parallel message passing hardware architecture?**

   Yes, this scheme achieved, approximately, a speedup of three for eight processors. Testing indicates that as the number of processors increases and the better the event interleaving becomes the greater the speedup.

   Advantages of this hybrid approach:

   - Incremental development possible - The conservative time synchronization protocol can be designed, implemented, and tested first. After successful testing for time synchronization, the optimistic lookahead processing can be developed.
   - Simulation integrity easy to determine and maintain - since the parallel implementation produces identical output to the sequential implementation, simulation integrity can be checked at any point in the execution. The output from the parallel implementation is ordered the same as the output from the sequential implementation.
   - Identical time events handled - This paradigm handles the case where identical time events must be executed in a particular sequence.
   - As the number of processors increases and the computation time increases the effectiveness of the scheme increases.
   - Load balancing - Since the goal of load balancing for this scheme is to provide good event interleaving, and event interleaving is based on event timestamps only, intuitively it seems that load balancing might prove easier in this case than in the case of battlefield partitioning. Although I haven't seen any studies on load balancing for a partitioned battlefield simulation, it would seem that dynamic resizing of battlefield partitions would be required together with

some knowledge of the state of the battlefield situation (i.e. Are many objects converging in battle ' to one logical process?).

- Battlefield Interactions - Although radio communications are not modeled in RIZSIM, it is clear to see that this hybrid design allows for complex battlefield interactions such as inter-object communications. Since the location of the object on the battlefield plays no role in the parallel protocol, an object on one end of the battlefield can send a radio communication message to an object at the other end of the battlefield. For example an AWACS controller can send flight instructions to an aircraft on the other side of a battlefield. This information would be included in the global update message transmitted following each safe event execution.
- A partitioned battlefield simulation that attempts to improve speedup by taking advantage of the distance between objects on the battlefield would have a difficult time maintaining that speedup while allowing near instantaneous cross battlefield radio messages.
- Application Specific Knowledge - Although some application specific know-ledge is required. The type of knowledge needed for this hybrid design relates more to knowledge of what types of data structures are used, and how and when they are updated.

Disadvantages of this hybrid approach:

- Data replicated across all logical processes - Data replication could limit the size of the simulation that can run on the parallel system. One of the advantages of partitioning the data is that a simulation too large to run on a single processor could be executed when spread across many processors.

2. **Is there an effective method to execute in correct sequence, identical time events on separate processors?**

Yes, tiebreaker information on the pending event for each logical process is maintained by each logical process. The tiebreaker is used to determine which event executes first in the case of identical time events.

3. **How much does the parallel programmer need to know about specific details of the simulation application in order to parallelize the simulation?**

   In this paradigm the programmer needs to know if identical time events must be executed in a specific order. If so, rule or rules for tiebreaking must be gleaned from the sequential simulation.

   Basically, the parallel programmer must understand which events cause what type of state changes in order to accurately reflect those changes in the update message. In RIZSIM an event execution results in state changes for one or two objects. Instead of sending the complete updated state of each object following each event execution, The protocol was designed to only send the changes to the states of those objects. Because of this, one needed to know what type of changes are made by certain events. For example, with RIZSIM, a Reached_Turnpoint event results in a routepoint being deleted from the routepoint list, whereas with an Entered_Sensor_Range event a routepoint is deleted and another routepoint is added. That added routepoint had to be included in the update message. This knowledge is essential for generating the contents of the update message.

   The parallel programmer must know where the computational complexity or intensity exists in the sequential simulation. The section of code that contains the computational intensity is what gets precomputed. Details of the algorithms are not required knowledge, but the parallel programmer must know the how the results are stored and how to update those results. This knowledge is essential for implementing the optimistic lookahead.

4. **How much does the application programmer need to know about parallel programming in order to facilitate parallelization of a sequential simulation?**

The application programmer doesn't need any knowledge of parallel programming techniques. However, the software design of the sequential simulation should make use of basic design principles such as functional cohesion and loose coupling in order to make the parallel programmer's job easier.

Ironically, in the interest of making the sequential program execute faster, the application programmer may "cut corners" in the design. For instance, in order to cut down on function calls a programmer might combine functionality into one large function or procedure. Although the sequential program may run faster, program understanding, software maintainability, and the ability to easily parallelize the simulation could be adversely impacted.

5. **How much of the parallel protocol design and code is reusable between different simulation applications?**

   Code Reuse - Some of the event list management functions and low level message passing functions are reusable. In general, for a battle simulation and the hybrid protocol some but not all of the code is reusable. The time synchronization and tiebreaking code could be easily reused. The state change update processing based on update message contents, and the lookahead and lookahead update processing are application specific and the code could not be easily reused.

   Design Reuse - The top level design concepts could be reused and tailored to fit other battle simulations. But because of some of the detailed knowledge required of the specific application it is difficult to judge how "easily" this paradigm and this design could be molded to fit other battle simulations.

6. **Is it possible to develop and use a simulation testbed that features interchangeable simulation applications and interchangeable parallel time synchronization protocols?** (related to #4 above)

   Spectrum purports to provide a framework to mix and match various applications and protocols. Spectrum provides functions that make calls to user supplied protocol specific functions (called filters) which take care of the parallel protocol processing.

The fact that these filters are application specific make it difficult to mix and match protocols. As a result I did not use the Spectrum "filter" mechanism.

It would be difficult to develop a testbed that coulu easily accommodate different battle simulations. Some parts common to all applications like the simulation driver might work in a testbed, but much of the complexity revolves around integrating the protocol with the particular application. The implementation of this protocol required some knowledge of the application, and that may very well be the case for every protocol and every battle simulation. The requirement for application specific knowledge and processing could defeat the ability of a testbed to be truly generic.

7.3 Recommendations/Further Research

7.3.1 **More Hybrid Research** Recommend further research in the area of hybrid protocols for discrete event battle simulations. Testing should be accomplished with larger simulation scenarios and larger number of logical processes or processors. Research in the area of load balancing of this type of protocol can be conducted by designing scenarios that result in unbalanced loads. Various heuristics and techniques can be tested to try to improve speedup by balancing the load.

7.3.2 **Perfect Event Interleaving** Recommend further research in attempting to acheive perfect event interleaving. One thought is to develop some sort of scheduling and distribution mechanism for determining when events will be executed on which processor.

7.3.3 **Combine Hybrid/Partitioned Battlefield** Recommend further research in combining this hybrid protocol with a battlefield partitioned simulation. Combining these two would hold the potential for realizing the speedup associated with a partitioned search space together with the speedup associated with the event lookahead.

7.3.4 **New AFIT Battle Simulation** Recommend developing a new AFIT battle simulation that makes use of object oriented or object based design techniques. This battle simulation should be flexible and modular so that modifications can easily made to in order to mimic the characteristics of different "real world" battle simulations. With this flexible, well designed battle simulation at the core, various protocols can be designed around it. In essence the battle simulation becomes the foundation for a testbed. Instead of mixing and matching applications and protocols as Spectrum attempts to do, this testbed would only support mixing protocols on a single application or variation of that application. Knowledge gained from this first step might lead to a better understanding of how to mix and match various applications and protocols.

7.3.5 **Parallel Object Oriented Databases** Recommend investigation into using parallel object oriented databases for discrete event simulation. Although this simulation used no database at all, many battle simulations use large databases for information such as terrain data. It is not beyond reason to believe that future simulations will need access to massive databases, and it would prove worthwhile to start investigating the database connection to discrete event simulation.

7.3.6 **Typical Battlefield Simulation** Recommend a study to determine what types of battle simulations are currently being used, and what are their major characteristics as they apply to parallel programming. This research should be conducted before the development of the AFIT battle simulation. Questions such as how objects move across the battlefield should be addressed. For instance do objects follow a prescripted plan of movement as in RIZSIM or does each object have some sort of built in decision making logic that controls movement based on objective and the current environment?

7.3.7 **Hypercube Model** Recommend investigating feasibility of developing a model of the Hypercube using a simulation language like SLAM. If the model is well designed then characteristics of a particular application and a particular protocol can be overlaid on the model. By characteristics I mean attributes such as message sizes, computational intensity, and message passing behavior. Speedup tests can then be conducted using the SLAM model without worrying about application specific details.

Appendix A. **RIZSIM Bugs**

A.1 Application Problems

A.1.1 **Subtraction Mixup.** subtraction calculation where the operands were on the wrong sides of a minus sign resulted in aircraft travelling in a direction opposite of what it should have been. This was fixed by moving the operands to the correct sides of the minus sign.

A.1.2 **Identical Sensor Range.** When two aircraft with identical sensor ranges encounter each other two identical routepoints are scheduled for each of these aircraft. An aircraft with consecutive identical routepoint locations will cause computation errors in the arctangent calculations for aircraft pitch. The scenario generator generates the same sensor range for all of the aircraft with the same loyalty. When two aircraft of the same loyalty sense each other a computation error will result. This problem was patched by inserting a conditional check for aircraft of the same loyalty in the sensor check function. If the aircraft are of the same loyalty then they ignore each other. This problem was patched and not fixed. If two aircraft have different loyalties but the same sensor range then the problem will re-occur.

A.1.3 **Stationary Object Sensing.** When an aircraft reaches its last routepoint it is essentially through with the simulation. It no longer moves and it no longer has sensing capability. It becomes a stationary object in space that can be sensed by other moving objects. When a moving object senses a stationary object a software failure occurs because an attempt is made to pop the next routepoint from the stationary object's empty routepoint list. This was fixed by putting in a check to see if the object being sensed is stationary by first checking to see if its routepoint list is empty.

A.1.4 **Past Event Scheduling.** When an aircraft gets destroyed then future events related to that aircraft must either be deleted, or deleted and rescheduled. RIZSIM was incorrectly rescheduling events which resulted in messages being scheduled at a time less than the current simulation time. The past time scheduling was the result of trying to reschedule events by executing sensor check without first updating position. This was fixed by simply rescheduling an event by changing it to a reached turnpoint event, and leaving it at its current position in the event list.

A.2 System Problems

A.2.1 **Floating Point Errors.** Occasionally a floating point equation would return as its result 'nan' (not a number). The cause of this problem is not known. The problem exhibited the following characteristics:

- The equations where this occurred always contained multiplication and addition on the same line of source code. $(x * x) + (x * x)$
- This simulation uses double floating point numbers in all calculations.
- The input variables to the equation were checked via *printf* statements during debugging and the input should have generated valid results.
- The input variables were local variables.
- The equation would give 'nan' results and then subsequently when the same line of code was executed again the results would be good. This indicates that erroneous pointer addressing was not corrupting that line of code in memory. If the memory location was corrupted then one would think that that instruction would give erroneous results for the rest of the program execution.

The workaround to the problem was to duplicate the line of code that sometimes gave 'nan' results and execute the identical line of code consecutively. The second execution of the identical instruction gave good results. Not every line of code in the simulation application involving multiplication and addition was replicated. Therefore the error may still be occuring on some of the larger scenario sizes. Tracing this error is very time consuming since it may not occur until well into a large scenario.

Appendix B. Results - Timing Runs

These tables show the raw results of the timing runs. Each table contains the scenario execution time in seconds for each scenario executing on various numbers of processors.

Table B.1. 64 Aircraft Scenarios - With Missiles

# Processors	1	2	4	8	# Events
Scenarios 64a	9.2	7.9	7.6	9.1	474
64b	9.2	8.1	7.6	8.0	508
64c	10.8	9.3	8.5	10.0	562
64d	9.9	8.5	8.0	9.8	511
64e	9.7	8.1	7.9	9.6	494
Ave. Speedup	0	1.16	1.23	1.05	
Ave. Use_Lookahead	0	.39	.5	.56	

Table B.2. 128 Aircraft Scenarios - With Missiles

# Processors	1	2	4	8	# Events
Scenarios 128a	35.1	27.2	22.7	22.2	1033
128b	35.3	27.7	22.9	22.6	1075
128c	35.6	27.7	22.6	22.0	1069
128d	37.6	30.0	25.1	24.2	1168
128e	36.7	28.7	23.6	22.9	1101
Ave. Speedup	0	1.27	1.53	1.58	
Ave. Use_Lookahead	0	.38	.52	.59	

Table B.3. 256 Aircraft Scenarios - With Missiles

# Processors	1	2	4	8	# Events
Scenarios 256a	150.2	114.8	92.4	77.2	2538
256b	157.5	120.2	95.7	79.6	2674
256c	148.4	115.8	94.2	70.3	2551
256d	138.7	104.9	83.3	75.8	2318
256e	151.3	114.5	89.9	75.8	2506
Ave. Speedup	0	1.31	1.63	1.97	
Ave. Use_lookahead	0	.38	.52	.59	

Table B.4. 64 Aircraft Scenarios - NO Missiles

# Processors	1	2	4	8	# Events
Scenarios 64a	15.8	12.2	10.2	9.5	659
64b	14.9	11.4	9.7	9.1	620
64c	15.7	12.0	10.0	9.6	649
64d	16.5	12.8	10.5	9.7	675
64e	15.7	11.9	10.2	9.9	652
Ave. Speedup	0	1.29	1.55	1.64	
Ave. Use_Lookahead	0	.47	.65	.74	

Table B.5. 128 Aircraft Scenarios - NO Missiles

# Processors	1	2	4	8	# Events
Scenarios 128a	86.6	60.2	44.8	35.9	1815
128b	89.9	62.2	45.7	36.6	1853
128c	87.4	61.4	45.6	36.5	1840
128d	93.1	64.7	47.6	38.0	1937
128e	93.1	64.8	47.1	37.0	1908
Ave. Speedup	0	1.43	1.94	2.44	
Ave. Use_Lookahead	0	.48	.67	.78	

Table B.6. 256 Aircraft Scenarios - NO Missiles

# Processors	1	2	4	8	# Events
Scenarios 256a	598.6	399.4	286.5	206.3	5945
256b	619.9	418.2	295.9	210.5	6231
256c	600.0	402.1	284.4	205.8	5984
256d	584.8	395.5	279.8	201.5	5885
256e	588.1	391.1	277.7	196.7	5916
Ave. Speedup	0	1.49	2.1	2.93	
Ave. Use_Lookahead	0	.47	.7	.8	

Table B.7. 512 Aircraft Scenario - NO Missiles

# Processors	1	2	4	8	# Events
Scenario 512a	4555.4	2953.5	2040	1482	21304
Ave. Speedup	0	1.54	2.23	3.07	
Ave. Use_Lookahead	0	.47	.69	.8	

Table B.8. 256c Scenario, Spin 10, NO Missiles

# Processors	1	2	4	8	# Events
Scenario 256c	743.0	499.6	352.7	254.6	5984
Ave. Speedup	0	1.48	2.1	2.91	
Ave. Use_Lookahead	0	.47	.69	.80	

Table B.9. 256c Scenario, Spin 100, NO Missiles

# Processors	1	2	4	8	# Events
Scenario 256c	2025	1374	965	696	5984
Ave. Speedup	0	1.47	2.09	2.90	
Ave. Use_lookahead	0	.47	.69	.80	

Table B.10. Perfect Interleave - Lookahead = Update

# Processors	1	8	# Events
Scenario 8	2723	683	40
Ave. Speedup	0	3.98	
Ave. Use_Lookahead	0	1.0	

Table B.11. Perfect Interleave - Lookahead time greater than Update time

# Processors	1	8	# Events
Scenario 8	2723	346	40
Ave. Speedup	0	7.8	
Ave. Use_Lookahead	0	1.0	

Table B.12. 256 Acft - Missiles - No Initial Interleave - Red/Blue Mix

# Processors	1	8	# Events
Scenarios 256a	150.2	97.8	2538
256b	157.5	100.8	2674
256c	148.4	100.5	2551
256d	138.7	89.4	2318
256e	151.3	95.6	2506
Ave. Speedup	0	1.54	
Ave. Use_Lookahead	0	.496	

Table B.13. 256 Acft - Missiles - No Initial Interleave - Red/Blue Separate

# Processors	1	8	# Events
Scenarios 256a	150.2	100.8	2538
256b	157.5	103.5	2674
256c	148.4	101.4	2551
256d	138.7	92.8	2318
256e	151.3	97.2	2506
Ave. Speedup	0	1.50	
Ave. Use_Lookahead	0	.488	

Table B.14. 256 Acft - No Missiles - No Initial Interleave - Red/Blue Mix

# Processors	1	8	# Events
Scenarios 256a	598.6	222.8	5945
256b	619.9	236.8	6231
256c	600.0	231.7	5984
256d	584.8	224.2	5885
256e	588.1	222.1	5916
Ave. Speedup	0	2.62	
Ave. Use_Lookahead	0	.758	

Table B.15. 256 Acft - No Missiles - No Initial Interleave - Red/Blue Separate

# Processors	1	8	# Events
Scenarios 256a	598.6	227.1	5945
256b	619.9	239.7	6231
256c	600.0	227.3	5984
256d	584.8	225.0	5885
256e	588.1	222.4	5916
Ave. Speedup	0	2.61	
Ave. USE_LOOKAHEAD	0	.768	

Table B.16. 256 Aircraft Scenarios - With Missiles - Psuedo Realistic

# Processors	1	8	# Events
Scenarios r256a	217.6	102.1	3294
r256b	211.1	99.9	3251
r256c	204.9	98.8	3181
Ave. Speedup	0	2.1	
Ave. Use_lookahead	0	.63	

Table B.17. 256 Aircraft Scenarios,Red/Blue Solid,With Missiles,Psuedo Realistic

# Processors	1	8	# Events
Scenarios r256a	217.6	127.0	3294
r256b	211.1	124.3	3251
r256c	204.9	121.6	3181
Ave. Speedup	0	2.1	
Ave. Use_lookahead	0	.55	

Table B.18. 256 Aircraft Scenarios - No Missiles - Various Routepoints

# of Processors	1	8	# Events	Ratio	Speedup
Scenarios 256 RP3	216.1	76.6	2444	.75	2.82
256 RP6	606.3	204.2	5980	.81	2.9
256 RP9	1139.0	371.3	10260	.82	3.0
256 RP12	1602.7	511.7	13740	.82	3.1

Appendix C. The Hybrid Files

File Descriptions:

OVERVIEW:

Application Files: See Rizza's thesis (29) for a description of these files:

sim_func.c
sim_func.h
sim_stru.h
events.c
events.h
ll.c

Simulation Driver Files:

rizsim.c - the original rizsim.c was replaced with another locally developed simulation driver. The replacement simulation driver retains the name rizsim.c as well as some modules that were in the original rizsim.c.

simdrive.c - contains the MAIN function in the format required by SPECTRUM.

dll.c - contains various doubly linked list management functions.

neq1.c - contains functions for managing the next event queue.

event.h - not to be confused with rizsim's events.h.

This file defines the following:

- contents of an event.
- conditional compilation variables to turn I/O on/off.
- global variables used for lookahead processing and lookahead update processing.

Parallel Protocol Files:

Spectrum Files:

- globals.h - defines the following:
  - structure of the update message.
  - structure of the final message sent to the host for statistical reporting.
  - input and output message buffer.
  - global_Q_number and tiebreaker array.
  - LOOP_COUNT for spin loops.

- application.h - of interest - defines the number of processors.

  This number must match the number of processors selected at runtime. After changing this number recompile.

- cube2.c - contains the functions that actually send

and receive messages. Three functions were
added:

- node_get_one_message.
- node_sned_one_message.
- node_send_last_time.

- host2.c - some code was added to facilitate termination processing and data reporting.
- lp_man.c - Only the initialization functions were used.
- myfilters.c - not used.

HYBRID FILES:

- rollback.c - described below.
- protocol.c - described below.
- rollback.h - described below.
- protocol.h - described below.

----

rollback.c

OVERVIEW: This file contains the modules needed to implement lookahead, lookahead update and local rollback.

GLOBAL VARIABLES:

Flags used to control execution of look_ahead and update:

USE_LOOKAHEAD

LOOKAHEAD_COMPLETE

MISSILE_FLAG

Temporary Storage Global Variables:

These two contain a copy of the pending event and the object associated with the pending event:

projected_event

projected_object

The following global variables contain data computed as a result of precomputing the next event:

missile_event

projected_new_routepoint

etype_projected

etime_projected

entity1_projected

entity2_projected

object1_projected

object2_projected

*note: defined in event.h

MODULES:

The following modules were essentially copied from RIZSIM. Changes were made so that when these modules execute a lookahead event the results are put into temporary storage.

```
projected_sensor_check
projected_attack
projected_update_position
projected_add_new_routpoint
add_event_coords_to_route_projected
projected_reached_turnpoint
projected_entered_sensor_range
projected_made_sensor_contact
projected_ordnance_released
ordnance_reached_target_projected
projected_collision_distance_reached
```

projected_init - Allocates temporary storage memory for global variables.

The following three copy modules makes copies of the info assciated with the pending event. Called by set_up_lookahead:

```
copy_object
copy_two_routepoints
copy_event
```

The following three execute the lookahead:

look_ahead

set_up_lookahead

projected_do_event

The following three update the results of a previous lookahead execution. The update is based on the content of an incoming update message. The incoming update message contains change information for a single aircraft, and perhaps a new missile. Therefore a single sensor_check is executed involving the aircraft associated with the incoming update message. The results of the previously computed lookahead calculations are either left alone, modified, or thrown out completely and new lookahead or a safe computation reexecuted:

lookahead_update

single_sensor_check

event_reprojected

The following module is executed when a lookahead and a lookahead update has completed and this LP is now the LP with the minimal next event time. This LP is now safe to execute and the lookahead results are made permanent.

make_lookahead_real

Debugging Modules:

show_object

display_event

show_update

show_projected_data

----

rollback.h - contains prototypes for rollback.c

----

protocol.c

OVERVIEW: The files in this module implement the conservative time

synchronization algorithm.

Global Variables:

```
               output_message_buffer
               input_message_buffer

               SEND_FLAG
               WAIT_ON_ISEND

               master_object_array
```

```
global_max_obj_id

USE_LOOKAHEAD
LOOKAHEAD_COMPLETE
MISSILE_FLAG
global_Q_number
tiebreaker
MSG_ID
```

The following module sets up memory for the one deep input and output message buffers:

```
init_msg_buffers
```

The following four modules handle processing incoming update messages. Before an event is executed incoming update messages (if any) are processed. Some other LP has safely executed an event and has sent update information including its new next event time, the new next event number or Q_number, whether or not it created a missile, and state changes to the aircraft associated with the event that it executed:

```
process_incoming_messages
process_message
update_attributes
create_a_missile
```

The following module determines if this LP has the global minimum pending next event time. It also handles tiebreaking if there are two LPs with same pending next event time:

```
          safe_to_execute
```

The following module is used to provide the pertinent aircraft state information in the outgoing update message following a safe event execution:

```
          build_message
```

The following module initiates an update message send to all LPs in the simulation. There are two versions: blocking send currently commented out and non_blocking send. With the non_blocking send a *WAIT_ON_ISEND* flag is set so that build_message does not overwrite the output message buffer while output transmission is still going on from a previous execution and send. Also, node_send_one_message must be adjusted depending upon blocking/non-blocking send.

```
          send_outgoing_messages
```

Debugging:

show_MOA

NOTE: some of the modules listed above might more appropriately fit into the application files i.e. sim_func.c since they contain application specific references and use the application's data structures. But since these modules were developed as part of this effort they were left in this file to make it easier to identify which modules were newly developed as part of this effort. Also sim_func.c is already a large file which takes a longer time to compile than protocol.c. Leaving them in this file facilitated quicker debugging due to quicker recompilation.

CONDITIONAL COMPILATION - OUTPUT turned on/off

There are conditional compilation 'defines' in the file event.h.

SCREEN_DISPLAY
GRAPHIC_DISPLAY

Comment them out and then recompile to turn off all I/O during a simulation timing run.
Leave them in to get screen display as output to the grapical file.of what is going on during the simulation.

SPIN LOOP LOCATIONS:

```
rollback.c - single_sensor_check

rollback.c - projected_sensor_check

sim_func.c - sensor_check


LOOP_COUNT - global variable in globals.h


---------------------------------------------------------------

---------------------------------------------------------------


MAPPING AIRCRAFT TO PROCESSORS:


Currently aircraft are mapped in a round robin fashion by the

function schedule_init_events in rizsim.c.  In conjunction with

this the tiebreaker array values (protocol.c) must initially match

the event_id or global_Q_number of the first event on each logical

process.  This method also assumes that the first line of the

input scenario contains aircraft #1, the second aircraft #2 and so

on.  Any other mappings will require changes to

schedule_init_events, the initial values in the tiebreaker array

and/or the ordering of the aircraft in the input scenario.


--------------------------------------------------------------

--------------------------------------------------------------


MAKEFILE:  The following is the HYBRID makefile.


CFLAGS = -O

SpecOBJS = cube2.o lp_man.o filters.o myfilters.o protocol.o
```

```
rollback.o
SpecSRC  = cube2.c lp_man.c filters.c myfilters.c protocol.c
rollback.c

HartOBJS = simdrive.o clock.o neq1.o event.o dll.o
HartSRC  = simdrive.c clock.c neq1.c event.c dll.c



RizOBJS  = events.o sim_func.o ll.o
RizSRC   = events.c sim_func.c ll.c

RizLIB = -lm

all: host rizsim

host: host2.c
        cc -o host -g host2.c -host

rizsim: $(RizOBJS) $(HartOBJS) $(SpecOBJS) rizsim.o
        cc -o rizsim $(RizOBJS) $(HartOBJS) $(SpecOBJS) $(RizLIB)
rizsim.o -node


rizsim.o: rizsim.c event.h
sim_func.o: sim_func.c event.h globals.h
ll.o: ll.c
events.o: events.c event.h
```

```
cube2.o: cube2.c globals.h cube2.h application.h
lp_man.o: lp_man.c globals.h application.h
myfilters.o: myfilters.c globals.h
filters.o: filters.c globals.h
protocol.o: protocol.c globals.h protocol.h
rollback.o: rollback.c  rollback.h event.h globals.h

simdrive.o: simdrive.c rollback.h
event.o: event.c
clock.o: clock.c
neq1.o: neq1.c event.h
dll.o: dll.c

--------------------------------------------------------------
--------------------------------------------------------------
```

PARALLEL RIZSIM EXECUTION:

Type "host" at the UNIX prompt while in the HYBRID directory.

Type "rizsim" in response to request for application name.

There are no command line arguments. Just hit return.

The number of cube nodes and the number of LP's should be the same. Other mappings have not been tested. Also, these numbers must match the NUM_PROCS defined in application.h.

```
Respond with "y" to natural node assignment in order to get one
logical process per node.
-------------------------------------------------------------
-------------------------------------------------------------


SAMPLE EXECUTION:


c386 42:host
Which application do you want to use?:rizsim
Enter the command line arguments for the program:
How many cube nodes do you want to use?:8
How many LP's are in this application?:8
Do you want to use the 'natural' node assignment?y
load -p 0 0 rizsim
load -p 0 1 rizsim
load -p 0 2 rizsim
load -p 0 3 rizsim
load -p 0 4 rizsim
load -p 0 5 rizsim
load -p 0 6 rizsim
load -p 0 7 rizsim
Cube Loaded
NODE: 0, Input Scenario Read_Time: 0.320000
NODE: 1, Input Scenario Read_Time: 0.340000
NODE: 2, Input Scenario Read_Time: 0.375000
NODE: 3, Input Scenario Read_Time: 0.382000
NODE: 4, Input Scenario Read_Time: 0.377000
```

```
NODE: 5, Input Scenario Read_Time: 0.368000

NODE: 6, Input Scenario Read_Time: 0.364000

NODE: 7, Input Scenario Read_Time: 0.422000

HOST: Wall clock time spent loading cube: 14 (secs)

HOST: Wall clock time spent waiting: 1 (secs)

NODE: 0, Total: 1.566000, Init: 0.412000, Sim: 14.785648, Do
Event: 7, Made Real: 15

NODE: 1, Total: 1.621000, Init: 0.444000, Sim: 17.142247, Do
Event: 9, Made Real: 11

NODE: 2, Total: 1.446000, Init: 0.531000, Sim: 10.909954, Do
Event: 4, Made Real: 15

NODE: 3, Total: 1.603000, Init: 0.602000, Sim: 19.394066, Do
Event: 6, Made Real: 16

NODE: 4, Total: 1.609000, Init: 0.636000, Sim: 23.022969, Do
Event: 5, Made Real: 13

NODE: 5, Total: 1.418000, Init: 0.666000, Sim: 9.985460, Do Event:
3, Made Real: 11

NODE: 6, Total: 1.617000, Init: 0.694000, Sim: 25.103488, Do
Event: 4, Made Real: 13

NODE: 7, Total: 1.613000, Init: 0.725000, Sim: 23.531224, Do
Event: 3, Made Real: 15

TOTAL DO: 41, TOTAL MADE: 109, TOTAL EVENTS: 150, USE_LOOKAHEAD
ratio: 0.726667

c386 43:


-------------------------------------------------------------


Total - at each node refers to total execution time.
```

Init - at each node refers to the initialization time.

Sim   - at each node refers to final simulation clock time on that node.

Do Event - at each node is the number of events executed normally.

Made Real - at each node refers to the number of events that USED LOOKAHEAD.

For the host the wall clock time spent waiting refers to the time the host spent waiting for all of the nodes to complete after all of the nodes were loaded.

```
----------------------------------------------------------------
----------------------------------------------------------------
```

Bibliography

1. Anderson, Lowell B. and others. *SIMTAX - A Taxonomy for Warfare Simulation*. Technical Report, Military Operations Research Society, 1987.
2. Bryant, R. E. *Simulation of Packet Communication Architecture Computer Systems*. Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.
3. Chandy, K. M. and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *5*:440–452 (September 1979).
4. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (May 1981).
5. Chandy, K. M. and R. Sherman. "The conditional event approach to distributed simulation." *Proceedings of the SCS Multi-conference on Distributed Simulation*. San Diego: Society for Computer Simulation, March 1989.
6. Dickens, Phillip M. and Paul F. Reynolds. "SRADS with Local Rollback." *Proceedings of the SCS Multi-conference on Distributed Simulation22*. San Diego: Society for Computer Simulation, January 1990.
7. Fujimoto, Richard M. "Parallel Discrete Event Simulation." *Proceedings of the 1989 Winter Simulation Conference*. New York: Association for Computing Machinery, December 1989.
8. Fujimoto, Richard M. "Performance measurements of distributed simulation strategies." *Proceedings of the SCS Multi-conference on Distributed Simulation21*. San Diego: Society for Computer Simlation, March 1989.
9. Gaffni, A. "Rollback Mechanisms for Optimistic Distributed Simulation Systems." *Proceedings of the SCS Multi-conference on Distributed Simulation*. 61–67. San Diego: Society for Computer Simulation, July 1988.
10. Government Accounting Office. *Advantages and Limitations of Computer Simulation in Decision Making*. Technical Report B-163074. Washington DC: Government Printing Office, May 1973.
11. Government Accounting Office. *DOD Simulations: Improved Assessment Procedures Would Increase the Credibility of Results*. Technical Report B-163074. Washington DC: Government Printing Office, December 1987.
12. Jefferson, David. "Preface." *Proceedings of the SCS Multiconference on Distributed Simulation*. San Diego: Society for Computer Simulation, 1988. Vol. 19, No. 3.
13. Jefferson, David and others. "Distributed simulation and Time Warp, Part 1: Design of Colliding Pucks." *Proceedings of the SCS Multiconference on Distributed Simulation*. San Diego: Society for Computer Simulation, 1988. Vol. 19, No. 3.
14. Jefferson, David M. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7:404–425 (July 1985).

15. Jefferson, David M. and others. "The Status of the Time Warp Operating System," *Communications of the ACM* (1988).

16. Lee, Capt Ann K. *An Empirical Study of Combining Communicating Processes in a Parallel Discrete Event Simulation*. MS thesis, AFIT/GCS/ENG/90D-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

17. Lubachevsky, B. "Efficient Distributed Event Driven Simulations of Multiple Loop Networks," *Communications of the ACM*, *32*:111–123 (January 1989).

18. Mannix, Capt David L. *Distributed Discrete-Event Simulation Using Variants of the Chandy- Misra Algorithm on the Intel Hypercube*. MS thesis, AFIT/GCS/ENG/88D-14, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

19. Misra, J. "Distributed Discrete Event Simulation," *ACM Computing Surveys*, *18*:39–65 (March 1986).

20. Nicol, David M. "Mapping a battlefield simulation onto message-passing parallel architectures." *Proceedings of the SCS Multi-conference on Computer Simulation*. San Diego: Society for Computer Simulation, February 1988.

21. Nicol, David M. "Parallel Discrete Event Simulation of FCFS Stochastic Queueing Networks," *ACM Sigplan Notices*, *23*:124–137 (September 1988).

22. Nicol, David M. *Performance Bounds on Parallel Self-Initiating Discrete Event*. Technical Report ICASE Report No. 90-21, Hampton, Virginia: Institute for Computer Applications in Science and Engineering, March 1990.

23. Nicol, David M. and Paul F. Reynolds Jr. "Problem Oriented Protocol Design." *Proceedings of 1984 Winter Simulation Conference*. New York: Communications of the ACM, December 1984.

24. Pritsker, Alan B. *Introduction to Simulation and SLAM II*. West Lafayette, Indiana: Systems Publishing Corporation, 1986.

25. Proicou, Capt Michael C. *A Distributed Kernel for Simulation of the VHSIC Hardware Description Language*. MS thesis, AFIT/GCS/ENG/89D-14, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

26. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." *Proceedings of the SCS Multi-conference on Distributed Simulation19*. February 1988.

27. Reynolds, Paul F. and others. "Comparative Analyses of Parallel Simulation Protocols." *Proceedings of the 1989 Winter Simulation Conference*. New York: Association for Computing Machinery, December 1989.

28. Reynolds, Paul F. Jr. "An Efficient Framework for Parallel Simulations." *Proceedings of the SCS Multi-conference on Computer Simulation*. San Diego: Society for Computer Simulation, 1991.

29. Rizza, Capt Robert J. *An Object-Oriented Military Simulation Baseline for Parallel Simulation Research*. MS thesis, AFIT/GCS/ENG/90D-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

30. Sokol, L. M. and others. "MTW: a strategy for scheduling discrete simulation events for concurrent execution." *Proceedings of the SCS Multi-conferenceon Distributed Simulation19*. San Diego: Society for Computer Simulation, jul 1988.

31. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm." *Proceedings of the SCS Multi-conference on Distributed Simulation21*. San Diego: Society for Computer Simulation, March 1989.

32. Wieland, Frederick and others. "Implementing a Distributed Combat Simulation on the Time Warp Operating System," *Communications of the ACM* (1988).

December 1991 Master's Thesis

A HYBRID APPROACH TO BATTLEFIELD PARALLEL DISCRETE EVENT SIMULATION

Steven R. Soderholm

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCS/ENG/91D-23

ASAS Project Office
SFAE-CC-INT-OR
1500 Planning Research Drive
Maclean, Va 22102-5099

Distribution Unlimited

This study describes a method of parallelizing a battlefield discrete event simulation. The method combines elements of conservative time synchronization together with elements of optimistic computation and local rollback on a message passing hardware architecture. The battle simulation features aircraft moving in a battle area and launching missiles at enemy aircraft. Aircraft are randomly grouped into logical processes (LPs), and a single LP is assigned to each processor. Aircraft state information is replicated across all LPs. Only the LP with the minimum next event time can execute safely. While one LP is executing safely all other LPs are precomputing their next event. When an LP does become safe to execute it can update its previous precomputation and broadcast the results to all other LPs. The sequential battlefield simulation has no battlefield partitions, and therefore the pros and cons of partitioning the battlefield in a conservative parallel implementation are discussed. Simulation speedup was achieved without battlefield partitioning and various simulation scenarios were run in order to investigate the impact of event interleaving among logical processes on simulation speedup.

Parallel Battlefield Simulation, Discrete Event Simulation 134

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave Blank)

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency. Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ..., To be published in .... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availablity Statement. Denote public availability or limitation. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR)

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- DOD - Leave blank
DOE	- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports
NASA	- NASA - Leave blank
NTIS	- NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.